SRE

Achieving Reliability with Boring Technology

DAVE MANGOT

Dave Mangot is the author of *Mastering DevOps* from Packt Publishing. Previously, he led site reliability engineering (SRE) for the SolarWinds cloud companies. An accomplished Systems Engineer with over 20 years' experience, he has held positions in various organizations, from small startups to multinational corporations such as Cable & Wireless and Salesforce, from Systems Administrator to Architect. He has led transformations at multiple companies in operational maturity and in a deeper adherence to DevOps thinking. He enjoys time spent as a mentor, speaker, and student to so many talented members of the community. usenix@mangot.com

Everything should be made as simple as possible, but no simpler.—*Albert Einstein*

Distributed systems. Complex systems. Enterprise systems. No matter how we're involved in computing these days, it's likely we're working on complex or complicated (in the Cynefin sense) problems. In fact, even systems that start out simple ultimately become complex through the continuing evolution of those systems through architecture changes, code deploys, or simply the passage of time (do you remember why you made that choice three years prior?). Because this complexity is a naturally occurring property of these systems, I choose to use boring technology.

When I say "boring technology," we should give credit to one of its biggest proponents, Dan McKinley (@mcfunley) who wrote: "We should generally pick the smallest set of tech that covers our problem domain, and lets us get the job done" [1].

Why do I feel this way? I've been doing Operations work for more than 20 years. I've worked in small startups and big multinationals. I've worked on huge monoliths and systems that had an undying allegiance to services. Through it all, I've encountered complexity. When it's 3 a.m. and the pager is blowing up, complexity is not my friend (or yours). Over the years, I've always tried to advocate for the "smallest set of tech that covers our problem domain." When you're firefighting and you're trying to reason about what is wrong, why the Java process keeps OOM'ing or why the database connection pool is being exhausted, the last thing you want is fancy, magical, technology.

MTTR > MTBF

You may be able to tell, I have a specific bias to the Operations perspective. As site reliability engineering (SRE) has become more prevalent, we can see an emphasis on reliability and recovery from failure. In an ideal world, our recovery from failure is instantaneous; the customer has no idea there was a failure. Unfortunately, we don't live in an ideal world, so the best we can do is to try to minimize downtime by maximizing our ability to recover from failure. Choosing boring technology is a proven technique for making this a reality.

Does your Operations (DevOps, SRE, etc.) really need to deploy multiple Kubernetes clusters in order to deploy a single Ruby script? Should we try to write our next service in Erlang because we heard it's "cool," even though our staff mostly consists of PHP programmers? Boring technology works well for us because we have more ability to reason about it. If my ability to form a mental model of the system I'm working on is hampered by my inability to understand the technology, either because of complexity or obscurity, I'm going to have a bad time.

Often the main problems with fancy technology is that it is optimized to try to prevent failures, not recover from failures. Many fancy "enterprise" technologies are created in this way. One way to think about this is in terms of horizontal vs. vertical scaling. You are probably in good shape if your solution is designed to scale horizontally, where the loss of any single component is easily handled by other easily replaceable components with no noticeable effect to the customer. If your solution is designed with multiple somethings (power supplies, network cards, etc.) within a single component, you may be relying on fancy technology. If you lose one of those systems, where does that leave you? Systems that optimize for mean time between failures (MTBF) instead of mean time to recovery (MTTR) are prone to what author Nasim Taleb calls "black swan events":

> [T]he problem with artificially suppressed volatility is not just that the system tends to become extremely fragile; it is that, at the same time, it exhibits no *visible* risks…These artificially constrained systems become prone to Black Swans. Such environments eventually experience massive blowups…catching everyone off guard and undoing years of stability or, in almost all cases, ending up far worse than they were in their initial volatile state. [2]

MTBF-Optimized Infrastructure

What are some examples of complexity evident in MTBF-optimized infrastructure? Have you ever configured network bonding on a Linux host? How many different modes are there for bonding? Six. That means that there are six different ways that you could possibly expect that your systems will behave in the event a network interface is lost. To what end? Well, to protect us from the case where a system could potentially disappear off the network. But is a NIC failure really the only way a system could disappear off the network? What about power supply failures? What about running out of memory or CPU? What about file system corruption? How many different components do we want to make redundant in order to guard against a system disappearing off the network? How much do we want to pay for those systems? Can we really foresee all possible failure scenarios?

What if we were to think about it a different way? What if we *expected* that systems would disappear off the network? If we design our systems in this way, we're *protected* from systems disappearing no matter *what* the reason! Additionally, because I'm spending less money per system, I can usually have more of them for the same cost. This increases my ability to tolerate failure, even *multiple* failures. This is another problem we often see when we try to choose fancy enterprise systems with multiple layers of complex protection within a single system. We can't afford many of the components, and thus we are often left with only two of something, a primary and a backup. Not only is this very inefficient (we've paid a lot of money for a system that most of the time does absolutely nothing), but in the event of a failure, we're now one failure away from catastrophic failure. Additionally, we're subject to relying on all that other money we spent on our enterprise support contract to deliver the necessary part on the 12x5 or 24x7 guaranteed response times as offered by the vendor. If the vendor doesn't have the part, or the power spike that blew out the first system comes back, we could be in a very bad situation.

Cattle vs. Pets

Instead, we should choose boring technology. If a system goes down, the load balancer stops sending it traffic because it's failed its health check, and we replace it with an exact replica. We don't care about an individual system, we care about the overall system. Many of you have probably heard of this as cattle vs. pets [3].

If a pet gets sick, we do what we can to make it better (like calling in enterprise support). If a head of cattle gets sick, we worry about the overall health of the herd. While we can't as readily replace one head of cattle, we can readily replace a server, especially in cloud or cloud-like environments.

As our systems mature and grow, we often see the wisdom of being able to control and reason about them in simple ways. This use of boring technology doesn't just have to apply to application servers, it can apply to networking or storage as well. Let's look at some examples.

Networking

If we were to look up the DNS information for www.atlassian.com (this is just one example), we would notice something interesting.

```
$ host www.atlassian.com
www.atlassian.com is an alias for pledge-vtm-ash2-prod
-public-01.atlassian.com.
pledge-vtm-ash2-prod-public-01.atlassian.com is an alias for
pe-vt-vtmnl-1h5icdrzt7xcp-d84e3144685e1b8d.elb.us-east-1
.amazonaws.com.
pe-vt-vtmnl-1h5icdrzt7xcp-d84e3144685e1b8d.elb.us-east-1
.amazonaws.com has address 18.234.32.152
pe-vt-vtmnl-1h5icdrzt7xcp-d84e3144685e1b8d.elb.us-east-1
.amazonaws.com has address 18.234.32.153
pe-vt-vtmnl-1h5icdrzt7xcp-d84e3144685e1b8d.elb.us-east-1
.amazonaws.com has address 18.234.32.154
```

Three IP addresses! That's strange! If you've ever spent any time with enterprise-grade networking gear, there is often a "floating IP" that can bounce back and forth between two pieces of equipment depending on which is currently responsible for handling the traffic (and the other sits idle, despite the fact that we've paid for it, just in case). That IP address would be presented to the world as a single IP. But in this case, we have three. Why? Because Amazon has the ability to replace components of its load balancers and actually does this with a fair amount of regularity. When they need to upgrade or replace a piece of hardware or software, they don't exercise the HSRP or VRRP sequence for shifting traffic to the "other" host. They replace the component itself, like cattle.

Storage

Solving a problem like storage at the level of Facebook could be a daunting challenge. If you needed to store all those baby pictures, profile pictures, wedding pictures, etc., that could be a tough problem. If you were Facebook, you may have started out using a number of enterprise class (or Pet) solutions. As a matter of fact, this was actually the case, until Haystack [4]. You can read the paper yourself, but this is from the conclusion: "Haystack provides a fault-tolerant and simple solution to photo storage at dramatically less cost and higher throughput than a traditional approach using NAS appliances. Furthermore, Haystack is incrementally scalable, a necessary quality as our users upload hundreds of millions of photos each week." Moving to a simple solution for the win.

Making Change

This idea of choosing simple (boring) solutions that we can reason about more easily may sound appealing at this point. But how do we make these changes in our existing organizations? How do we get to a point where we have simple recovery that we know both works and is well tested and practiced? As Gene Kim says of DevOps in "The Three Ways" [5], "repetition and practice is the prerequisite to mastery."

Just as Facebook was happy that their solution was incrementally scalable, the happiest path to making these kinds of changes is incremental as well. While we'd all love to have Netflix's Chaos Monkey running in our infrastructure tomorrow, proving all is well, that's as unrealistic as standing up a shiny new Kubernetes cluster tomorrow and understanding how to deploy and operate it. My favorite method for making change is what we often call *Crawl-Walk-Run*.

Crawl-Walk-Run

We are not born with the ability to run. There is a progression we must go through in order to reach that level of mastery (which takes repetition and practice!). So it is with maturation of processes or architecture when we are adopting boring technology.

Crawl

So how do we get started? How can we "crawl" when moving from our fancy enterprise technologies to something simpler? The first step is to configure just about everything with code. When we say everything, we mean Docker containers, servers, network gear, RAID cards, etc. We are trying to configure everything this way. This gives us a number of advantages:

- If we're doing infrastructure as code, we can version things, because they are in revision control. That means if I ever want to know how something was configured on March 22nd, I can look that up.
- That ability also gives me the ability to create representative test environments and have confidence that those environments are configured in the same way as production. If my test fails in a representative test environment, I have high confidence it would have failed in production.
- I also have confidence that any time I have a component of type X, it will be configured identically to every other component of type X with the "push of a button." One need look no further than the Knight Capital failure [6] to recognize the dangers of having differently configured systems that are supposed to be identical. Reasoning about multiple possible configurations of the same component interacting with each other is extremely difficult! Remember our Amazon load balancer example? Every time a load balancer component is swapped out, Amazon knows exactly how the new component will be configured. Every time a new Haystack node is deployed at Facebook, they know exactly how it will be configured.

There are many ways to configure things as code. We have configuration management tools, and we have config files or settings that can be checked into repositories. We can even use things like Puppet types and providers to interact with our RAID cards or out-of-band management cards to make sure they are configured perfectly every time. Many network vendors are now offering APIs we can interact with for our network gear to make sure they are configured properly.

If your fancy piece of tech does not offer a programmatic way of configuration, you are probably not using boring technology and have something designed to be manipulated by the messy bags of mostly water we call humans. Eliminate those from your infrastructure—the component, not the humans!

Walk

Now that we have confidence that our infrastructure will be configured properly each and every time (how quickly could you rebuild a server that was removed with an exact replica?), we are ready to experiment with failure. One relatively easy way to accomplish this is with production readiness game days.

In this scenario, before we allow a new service or major infrastructure component to be deployed to production, we test it to learn about failure. How does it fail? What is impacted? How do we even know it's failed? How do we recover?

If repetition and practice are the prerequisite to mastery, then we need to have an opportunity for repetition and practice. We do this by making a test plan of exactly what we will fail (in our representative test environment) and what the expected behavior will be. Maybe we will block the DNS servers. Maybe we will pull a disk. Maybe we will terminate an instance. There are many options. We also need to determine where the test data will come from if required. Copying production data can have security implications. Can we use synthetic data? This plan should be agreed upon by all the parties involved (Dev, Ops, DBA, etc.). Then the plan should be executed. This has a number of advantages:

- No complex systems can ever be "thrown over the wall" to Operations for deployment. If there is an unexpected behavior during the failure scenarios, the party responsible for fixing that behavior will be given as many opportunities as necessary to fix the offending behavior until the game day is declared successful.
- The folks responsible for remediating failure will have the opportunity to practice those remediations! No one wants the first time they attempt to recover a failed system to also be the first time anyone has ever attempted to recover said system. By practicing before production, you have the opportunity to not only learn how to do it, but to also ask for clarification, make suggestions, improve documentation, etc.
- We can often discover unintended consequences of the deployment of the new system. This is why representative test environments are so important. We don't want to discover that our database would run out of connections the first time the system is activated in production.
- It reinforces the idea that the availability of our production systems is everybody's responsibility. Not just the people who will be woken up in the middle of the night, but the entire team responsible for delivery of that component of the infrastructure.
- It gives us an opportunity to find out where our technology is not boring. If, during the game days, we repeatedly have problems restoring our systems to the proper state, or understanding the failure scenarios, maybe our system is not quite as boring as we thought. That is an opportunity to revisit the design, and the choices made, and make the necessary adjustments so that we can eliminate single points of failure, fancy vendor solutions that never quite live up to their promise, or that configuration that everyone could have sworn was in revision control but in fact was only placed as an unintended side effect of some other process.

Run

Once we've settled on our boring technology, and have confidence in our infrastructure and ability to detect and remediate failures, it's time to make that a regular part of how we operate. Both in participating more regularly in the design phase as well as after the system is deployed. This is a great time to get started with chaos engineering, a natural progression from the use of boring technology.

As Nora Jones said at ReDeploy 2018, "Chaos Engineering isn't done to cause problems; it is done to reveal them" [7]. We already know that our systems become more complex over time and that the system that we deployed two years ago has changed or morphed over time into something that can have many different properties than it did when first deployed. How do we ensure we can still recover from failures? By continually testing the infrastructure to make sure that the result of failures continues to be as we expect.

The problems that we will experience in production will become problems because complexity is an emergent property of these systems. If we expose those problems under controlled circumstances (people in the office at their desks, only one variable changed at a time, etc.), we will have a much higher likelihood of being able to detect and recover quickly, and then work to prevent those problems in the future. If we have these problems but don't reveal them, then we are setting ourselves up for Taleb's black swan events that can "catch everyone off guard" and "undo years of stability." That doesn't sound very boring to me!

Conclusion

When working in our professional roles as SREs, or storage administrators, or network engineers, etc., we are often heavily invested in the technology choices we make. Sometimes we may want to use some new technology because it's got a great reputation or because a lot of other people are using it. If it is not a technology that we understand well, or have the ability to understand well, we can often make choices that will cause us more problems down the road.

For that reason, when facing these choices, it is good to remember to choose boring technology. The complexity will be there, there is no running away from that. The systems will grow more and more complicated until it's time for that big refactor, which is a recognition that our systems are no longer boring but, rather, are collapsing under their own weight of complexity.

But there are ways to minimize those conditions and for us to mature our way out of bad situations when we find that we are in one. Choose boring technology.

References

[1] http://boringtechnology.club/.

[2] N. Taleb, *Antifragile: Things That Gain from Disorder* (Random House, 2012), p. 105.

[3] R. Bias, "The History of Pets vs Cattle and How to Use the Analogy Properly," September 29, 2016: http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/.

[4] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, "Finding a Needle in Haystack: Facebook's Photo Storage," 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10): https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Beaver.pdf.

[5] G. Kim, "The Three Ways: The Principles Underpinning DevOps," 2012: https://itrevolution.com/the-three-ways-principles-underpinning-devops/.

[6] D. Seven, "Knightmare: A DevOps Cautionary Tale": https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/.

[7] N. Jones, "Chaos Engineering: A Step Towards Resilience": https://youtu.be/qyzymLlj9ag?t=399.