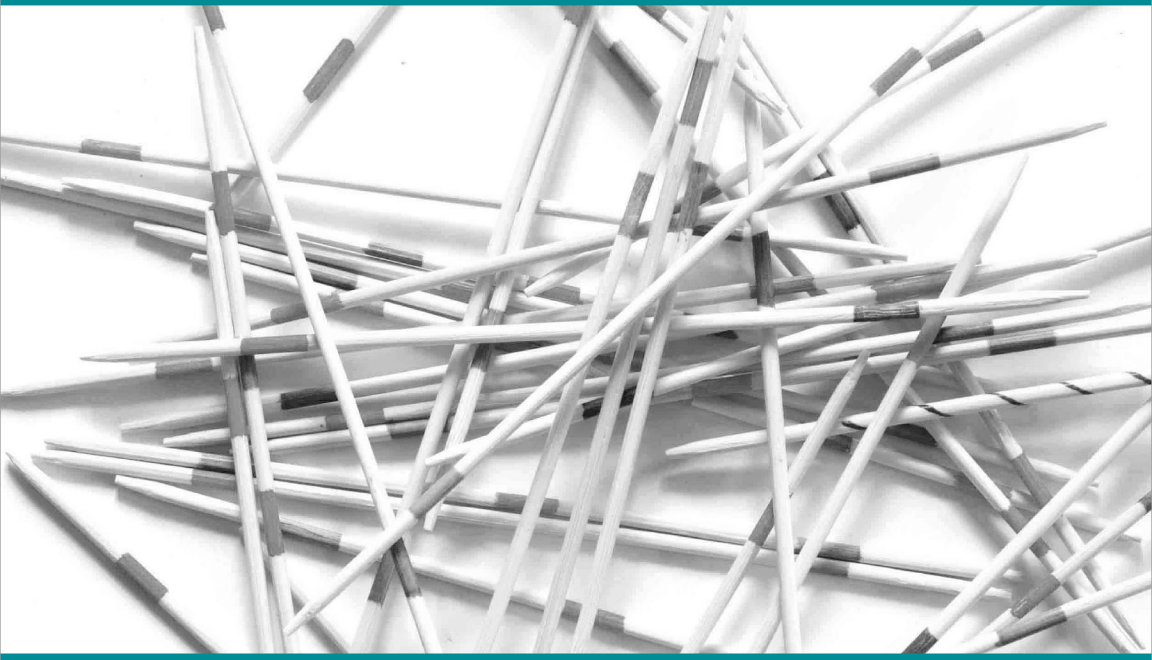
O'REILLY®

Learning from First Responders

When Your Systems Have to Work



Dylan Richard

Learning from First Responders: When Your Systems Have to Work

Dylan Richard

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Learning from First Responders: When Your Systems Have to Work

by Dylan Richard

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

February 2013: First Edition

Revision History for the First Edition:

2013-03-04: First release

2015-03-24: Second release

ISBN: 978-1-449-36414-4

Table of Contents

Introduction

In early summer of 2011 a group of technologists was assembled to build infrastructure and applications to help reelect The President of the United States of America. At the onset, the technology team at Obama for America (OFA) had just shy of 18 months to bring together a team of 40 odd technologists, define the technical direction, build out the infrastructure, and develop and deploy hundreds of applications. These applications all needed to be in place as soon as possible and would be vital to helping the campaign organize in some way.

With that in mind we pulled together a team of amazing engineers (most with non-political backgrounds) and started the process of building on top of what the previous presidential cycle had started. In large part the main task was to refactor an existing infrastructure in order to unify several disparate vendor applications into a well-defined and consistent application programming interface (API) that could enable brand new applications. These were as disparate as new vendor integrations to the tools that would help our more than 750,000 volunteers organize, fundraise, and talk directly to voters (along with hundreds of other backend and user-facing applications) to be built on top of it. In addition to building the core API, we also had to actually build the aforementioned applications, most of which were expected to scale to sustained traffic on the order of thousands of requests per second. All this while an organization the size of a fortune 500 company is being built up around you and needs to use those applications to help it grow.

No big deal.

All of this needed to be functional as soon as possible because at the scale of volunteering we were talking about, transitioning people to a

new tool too late in the game was not a realistic option, no matter how clean it was coded or how nicely designed. And so, working with our product management team and the rest of the campaign we spent the lion's share of the next year and a half building out the basic infrastructure and all features that would allow organizers and other staff to benefit from these applications.

As election day neared, it became clear that the focus the team had maintained for the "building" phase of the campaign needed to change. Performance and features had been the guiding principles up to this point; stability had been an important, but secondary goal. But for us to truly succeed on the four days that mattered most, we had to focus our efforts on making our applications and core infrastructure rock-solid in ways that the team had not fathomed before.

18 Months to Build Software that Absolutely Has to Work for 4 Days

With the team's focus so much on producing the software, adding features, and engineering for scale, a culture and way of working formed organically that was generally functioning, but had a couple of major flaws. We were finding ourselves shipping certain products (like the core of our API) and ignoring others (like the underpinnings for what would become our targeted Facebook sharing). This was happening without a grander view of what was important and without a focus on things that we as engineers found compelling, sacrificing other important products that needed to be built. In an effort to address these flaws while maintaining the agility and overall productivity, small teams were formed that focused on single workstreams.

As they were asked to do, these teams put their focus on servicing the needs of their workstream and only their workstream. Dividing the labor in this way ensured that we were not sacrificing functionality that would be incredibly important to one department in order to go deeper on other functionality for another department. In other words, it allowed us to service more widely rather than incredibly deeply.

The unfortunate flip side of this division of labor was an attitudinal shift away from everyone working together toward a single goal and toward servicing each workstream in a vacuum. This shift manifested itself in increased pain and frustration in integration, decreased intra-team communication, and increased team fracturing. Having that level of fracturing when we were metaphorically attempting to rebuild an airplane mid-flight was an incredible cause for concern. This fracturing grew over time, and after about a year it got to the point that it

forced us to question the decision of dividing and conquering and left us searching for ways to help the various teams work together more smoothly.

A fortuitous dinner with Harper Reed (the Chief Technology Officer at OFA) and a friend (Marc Hedlund) led to a discussion about the unique problems a campaign faces (most startups aren't dealing with arcane election laws) and the more common issues that you'd find in any engineering team.

While discussing team fracturing, insanely long hours, and focusing on the right thing, Marc suggested organizing a "game day" as a way to bring the team together. It would give the individual teams a well-needed shared focus and also allow everyone to have a bit of fun.

This plan struck an immediate chord. When Harper and I had worked at Threadless, large sales were planned each quarter. The time between each sale was spent refactoring and adding new features. A sale would give the team a laser focus on the things that were the most important, in this order: keep the servers up, take money, and let people find things to buy. Having that hard deadline with a defined desired outcome always helped the engineers put their specific tasks in context of the greater goal and also helped the business stakeholders prioritize functionality and define their core needs.

It also dovetailed nicely with the impending election. We were about two months out from the ultimate test of the functionality and infrastructure we had been building for all of these months. Over that time we had some scares. A couple of unplanned incidents showed us some of the limitations of our systems. The engineers had been diligent at addressing these failures by incorporating the failures into our unit tests and making small architectural changes to eliminate these points of failure. However, we knew that we had only seen the tip of the iceberg in terms of the kinds of scale and punishment our systems and applications would encounter.

We knew that we needed to be better prepared — both to know what could fail but also what that failure looks like, what to do in case of failure, and to make sure that we as a team could deal with that failure.

If we could do this game day right, we could touch and improve on a bunch of these issues and at the very least have a ton of fun doing it. With that in mind, we set out to do the game day as soon as possible. By this point we were about six weeks before election day.

Planning

Moving election day back a bit to accommodate this plan was a non-starter, as you might imagine. A concerted and hasty effort had to be organized to pull off a game day only a month prior to the election. The management team dedicated time to trading horses and smoothing feathers to buy the engineering team time to be able to prepare for failure.

The eventual compromise that was reached for buying that preparation time was finishing up another couple of weeks of features followed by a considerable (for the campaign) two-week feature freeze to code around failure states. This was done both to keep the engineers on their toes as well as keep the teams from shifting their focus too early. Engineers weren't told about the game day until about two weeks before it would take place — in other words, not until the feature freeze time were the teams spending their effort on preparation.

The teams were informed on October 2nd that game day would take place on the 19th of October. The teams had 17 days to failure-proof the software they had been building for 16 months. There was absolutely no way the teams could failure proof everything, and that was a good thing. If it wasn't absolutely important for our "Get Out The Vote" efforts, the application should fail back to a simple read-only version or repoint the DNS to the homepage and call it a day. Because of this hard deadline, the teams did not waste time solving problems that didn't matter to the core functionality of their applications.

Lead engineers, management, and project managers met with stakeholders and put together a list of the applications that needed to be covered and more importantly, the specific functionality that needed to be covered in each. Any scope creep that may have been pervasive

in the initial development easily went by the wayside when talking about what absolutely needed to work in an emergency situation. In a way, this strict timeline forced the management into a situation where the infrastructure's "hierarchy of needs" was defined and allowed everyone involved to relentlessly focus on those needs during the exercise. In a normal organization where time is more or less infinite, it's very easy to attempt to bite off too much work or get bogged down in endless minutiae during this process. Consider imposing strict deadlines on a game day to enforce this level of focus.

For example, features that motivate and build communities around phone banking were incredibly important and a vital piece to the growth of the OFA Call Tool. However, those same features could easily be shed in an emergency if it meant that people could continue to actually make phone calls — the core purpose of the application. While each application's core functionality had to be identified and made failure resistant, it was also beneficial to define exactly which features could be shed gracefully during a failure event.

The feature set hierarchy of needs was compared against our infrastructure to determine what pieces of the infrastructure it relied on, and how reliable that was deemed. For example, given a function that relied on writing information to a database, we would have to first determine how reliable that database was.

In our case, we used Amazon Relational Database Service (RDS) which took a lot of the simple database failure worries out of the way. However, there were still plenty of ways that we could run into database problems — replicant failures could force all traffic to the master and overrun it, endpoints that aren't optimized could be exercised at rates we had never seen, RDS itself could have issues or worse, EBS (Amazon's Elastic Block Storage) could have issues, we could have scaled the API so high that we would exhaust connections to the database.

With that many possible paths to failure, it would be considered risky, so we would either need an alternate write path or to find a way to get agreement on that functionality being non-essential. In our case, we relied heavily on Amazon Simple Queue Service (SQS) for delayed writes in our applications, so this would be a reasonable first approach. If the data being written was something that could not be queued or where queueing would introduce enough confusion that it would be more prudent to not write (password changes fell into that category), those features were simply disabled during the outage.

Alternately, given a function that needed to read data, we would go through the same assessment, determine that the database is risky and asses the options for fallbacks. In our case that usually meant caching in Amazon ElastiCache on the API side as well as in each of the clients. The dual caches and the database fallback was together a pretty reliable setup, but on the off chance that the database and the caches failed, we would be stuck. At this point, we would either need to determine other fallbacks (reading out of a static file in S3, a completely different region) or determine if this was a viable failure point for this function.

The teams spent a frantic two weeks implementing fail-safes for as many core features as possible. The service-based architecture that backed nearly every application allowed for this to be a much simpler task than it would have been with monolithic applications. Constraining the abstraction of failures in downstream applications and infrastructures to the API layer made it so that attention could be concentrated almost entirely in a single project. The major benefit of this was that applications higher in the stack could focus more on problems within their own domain rather than duplicate efforts on complex and tedious problems.

As this frantic sprint came to an end, the engineers had made some rather extreme but simple changes to how failures were handled that should allow for various dependencies to fail without hurting everything. People were still putting finishing touches on their changes when they learned in a team meeting that game day would take place on Sunday instead of Friday. That was the only information the engineers received. Management was purposefully vague about the particulars to avoid "studying for the test" rather than learning the material, as it were.

On Saturday, the team was sent a schedule for the game day. The schedule outlined the different failures that would be simulated, in what order, what the response plan should be, and when each test would begin and end.

Almost everything in the email was a lie.

Starting the Exercise

The first thing on the schedule for Sunday was to validate the staging environment to make sure that it was as close to production as possible. We had maintained a staging environment that was functionally equivalent to production, but generally with a smaller base state (single availability zone rather than three, and sometimes smaller box baselines) that we used for final integration testing, infrastructure shakeout, and load testing.

In this case, we decided to use it as a viable stand-in for production by beefing it up to production-like scale and simulating a bit of load with some small scripts. All of our applications had been built to scale horizontally from the beginning and had been rather extensively load tested in previous tests. With that in mind we launched a couple of bash scripts that simulated enough load to simulate light usage, not full scale. Having some load was important as much of our logging and alerting was based on actual failures that would only come with actual use. While it would be ideal to run a test like this in production, we were testing rather drastic failures that could have incredible front-facing effects if anything should fail unexpectedly. Seeing as failure had not yet been tested, any and all failures would fail unexpectedly. Given this, and that we were talking about the website of the President of the United States of America, we decided to go with the extremely safe approximation of production.

While the engineers were validating the staging environment, Nick Hatch on our devops team and I worked to set up our own backchannel to what was going to be happening. As the orchestrators of the failures, we needed a venue to document the changes that we would be making that would be inflicting the failures on the engineers.

In addition to the backchannel, we (the devops team and I) decided that since we were attempting to keep this as close to what a real incident would be like, and since we were all nerds, that we should essentially live action role play (LARP) the entire exercise. The devops team would be simultaneously causing the destruction as well as helping the engineers through it. It was vital to the success of the exercise that the devops team have split personalities, that they not let what we were actually doing leak through, and instead work through normal detection with the engineers without that knowledge.

One thing to be expected in a game day is that you cannot expect it to go according to plan, even for those who are planning it. The organizers of the event and the individuals participating in it should be ready and willing to go with the flow and make adjustments on the fly. In fact, just as the engineers were sending the final tags to staging and the plan set with Nick Hatch on how to begin the game (the first "issue" that would occur would be loss of all the database replicants, a supposed no-op) when all of a sudden reports began to trickle in about legitimate issues downstream that were affecting the payment processor.

The OFA Incident Response Campfire chat room was suddenly a cacophony of engineers wondering if this was really happening or a part of the test. In an effort to keep game day as close to a real incident as possible, we intended to use our actual Incident Response channel for real-time communication. With real incidents impinging on game day, it became clear that this was not going to work — the decision was made to separate the simulated incident response discussion into its own channel on the fly and the tech finance team would have to fight both the production issue as well as any game-day issues at the same time. After all, when it rains it pours.

With adjustments to the plan made, Hatch changed the security group on the database replicas and as far as all of the code was concerned, the replicas were down. During this action and from there on out, no communication was made with the engineers about what was happening. The organizers of the game day sat back and watched the incident response channel to see if the engineers would notice that they were operating with only a single master now and how long it would take. It took about four minutes.

Four minutes to recognize a failure event while the engineers were anticipating failures is way too long. Four minutes while engineers are

paying attention is 15 minutes or more when no one is looking. If it's a failure condition, you should alert on it. If it's a condition of note, you should measure it. If you're measuring it using statsd and Graphite you can alert using Seyren. The teams were using all of these tools, but they weren't properly alerting on many common failure cases.

At this point it was clear that there were legitimate process and alerting issues inherent to our infrastructure. Engineers asked the organizers to pause the event so logs could be analyzed to ensure that all of the right spots had been hit and the new code was acting as expected. Well, this was a simulation of real life, and real life doesn't stop and wait. Real life says that if you lose your replicants and move all reads to your master, your master is probably going to die. That's precisely what happened next.

Real Breaking

This time, it didn't take four minutes to notice. Almost immediately, all of the applications failed. Game day had gone off-script and the engineers suddenly realized it. The API team had spent the previous two weeks ensuring that the software could handle a master dying by seamlessly failing into a read-only state. And yet, as soon as the master failed, our entire infrastructure went with it.

The master database failure was handled exactly as it should have been by the core API, but the identity service used the database directly (oh technical debt, you take so little time to be costly). An immediate failure spread across all of our applications, instead of the planned switchover to read-only. This was about the point where the line between reality and FAILARP-ing (live action role playing) really started to blur. Systems failing in unexpected ways brings out a visceral mixture of fear, anger, and futility that a real incident brings about.

In the backchannel, we decided to bring the master back up. Identity was such a core service that there was no point in testing anything beyond what we had tested with the master down. We noted that certain clients would need to handle a downed identity server in a better way, and that our identity server would need to grow far more fault tolerant in the immediate future. The feeling of preparedness was gone.

Rather than just fix the permission group on the master, we promoted a replicant so we could see in a controlled environment what that looked like, and again to better simulate reality; sometimes databases that die stay dead. As it turned out, if we hadn't tested it that way we wouldn't have known that promoting a replicant on RDS breaks replication immediately and leaves you with just a single master that you

then need to stress by taking a backup to create a replicant. Information that it turns out is incredibly important to have; if you know that this occurs, you can disable endpoint to shed load to make sure that you don't immediately kill your shiny new master.

We went on to simulate several other scenarios that mimicked real-life failures we had seen before: having replicants flap available and become unavailable (to simulate this we would just revoke access to them, then reinstate access), break your caching layer, simulate full disks (revoke write privileges), and some human error. The easiest human error simulation is to just not do a piece of a process but say that you did: for example, starting replication.

In a four-to-five-hour torrent of breaking things, we had worked through our list of things we would be testing and everyone was exhausted. In the game day incident response channel, we told the team that their nightmare was over, thanked everyone and asked them to gather their notes and compile them per workstream. At the same time in the backchannel, we decided to test the fallbacks. Almost all of our fallbacks for database failure relied on writing to an SQS queue for later processing. SQS for many things was what we relied on in case of failure.

So we killed it.

One or two of our stacks were deliberately set up to be decoupled and wrote to a series of SQS queues that would then get written to the database in a separate process. They had handled all of the tests that we had thrown at them that day because they had been built from day one to be able to operate without a database. So we turned off SQS (in reality, what we did was to modify the /etc/hosts on all of the boxes to point the queue URL to a nonsense URL). Even given the "all clear," there was appropriate logging and alerting in place, and our final deceitful breakage of the day was found nearly immediately.

In reality there was not much that we could do in the case of an SQS failure, but we now knew how the applications reacted to it and could fall back to logging or otherwise staging the data write if need be.

Reflections

We worked as a team across all the workstreams to make sure that we stayed available in every instance. We found ways for people who were generally not affected by outages to be the first line of defense for external queries, allowing everyone to own the process. We re-learned that in the end, most of handling an outage is about communication, both internal to the technical group as well as to the stakeholders, and most importantly to the users. As a team-building exercise, the shared burden was in my eyes a wonderful success.

There are technical failures and process failures, and in testing the former, we found how to improve on the latter.

Forcing a timeline meant that we were able to focus on the things that matter. In our case, we had a deadline that couldn't possibly move. However, that wasn't the deadline that actually mattered for defining that focus; the game day date was the real cut off. Manufactured deadlines can be just as effective at giving the focus, but some relation to reality is helpful. If you find yourself getting bogged down trying to failure proof 100% of your application, you'll never get there. Do monthly game days and actively choose a small piece to have failure modes each time. Think of it this way, if you bite off 20% failure coverage for each game day, you'll be in infinitely better shape than trying to failure proof everything before you test it.

Responding to these simulated and legitimate incidents helped us in innumerable ways, but perhaps the most valuable lesson was how it solidified the mechanics of our incident response process. An initial draft of a document on how incidents should be handled was written in June, 2011. However, it wasn't until we had a mature team working

through actual incidents that we were able to bridge that document to a process that worked for our team.

There were a couple of takeaways to the process side of dealing with an incident that are worth highlighting.

Having a defined chain of command is something that was absolutely necessary to getting anything done. Having an influx of a dozen people who are all smart and all have great ideas trying to steer a ship in a dozen directions is a recipe for disaster. There needs to be someone who can listen to people and say "Yes, this is what we are going to do." Establish upfront who that person is.

Having one person who kept a running shared document (Google Docs works great for this) of the incident meant that communication to stakeholders was as easy as pointing them to a web link and answering whatever questions they had about it. Having that document also meant that as people joined the channel they were able to catch up on what the situation was without having to interrupt current problem solving happening in channel. Protip: change the subject of the room to what the current incident is, and add a link to the current incident document.

We also found some flaws in our process that we didn't have viable solutions for. For instance, we would regularly have several different incidents that needed to happen at the same time. (Hilariously, less so during the game day. It turns out that as sadistic as I was, I was nicer than reality). As soon as there is a second incident, communication in a single channel would become overly confusing. Our work-around for this was to move the discussion for secondary incidents to other rooms that made sense, and to have someone (usually me) act as the dispatcher who would redirect conversations to the appropriate rooms. Not a scalable system, but we were able to make it through.

In addition, the results of the failures were documented per application and runbooks were generated. Runbooks are essentially incredibly simple "if this, then that" instructions for assorted ifs. We bundled these runbooks with the documentation and READMEs in the git repositories for the respective applications so that anyone responding to an incident could know how to respond.

In the future, I will use game days to inform the incident response process, not the other way around. Accepting that there are different responses for each application is something that needs to happen.

While a baseline set of guidelines for responding to any incident is necessary, we also need a runbook for each application.

On the technical side, we were able to validate that many of our assumptions were correct and learned that some of them were incredibly far off. Our assumption that using a service-oriented architecture would mean that we could isolate cross-application issues to that layer was demonstrably false, hoisted by our own technical debt.

The most obvious take away from the game day was identifying where exactly we failed unexpectedly and where we had more to work on. We took this as direction on where to focus efforts for the next sprint and retested. We ended up doing a scaled back version of the game day nearly every week for the following three weeks to validate that we were getting more and more coverage in our failsafes.

The often repeated adage of "if you don't measure it, it didn't happen" bore true, but also led us to the corollary of "if you don't alert, it doesn't matter if you measure." We learned that there was great benefit in measuring both success and failure and that alerting on the absence of success could find problems that only alerting on failure would miss.

As far as failovers, we found our basic structure was sound. For databases, on slave failure shift seamlessly to master; on master failure, fall back to slaves and fail to read only (or other fallbacks above the database level, like queuing writes where appropriate); and on failure of both master and replicants, fail to cache. This isn't really novel in any way, and that's the point. Having clearly defined failure modes was far more successful than trying to be clever and having it all appear to be working.

Having the API return headers explaining the health of the API was not necessarily obvious, but it was an incredibly helpful way to make sure that clients were able to do any higher level failure handling.

Feature flags that are configurable proved to be an incredibly important tool in our toolset. While simulating load, we pushed on a single endpoint that had very expensive queries, which had incredibly adverse effects on the database, specifically pegging its CPU usage. In an emergency like that, the ability to turn off a single endpoint instead of failing over the entire data infrastructure is invaluable. Our game day validated that some of our initial choices about how to implement those switches were sound. A first draft at implementing feature flags had the flags set in a configuration file in the code. Changing the flag

would necessitate a full deploy. We switched to a simple flat file in Amazon's S3 that would be cached in memcache for a set number of minutes with a known key. Having it be a simple file was an important decision. Being able to make the changes with only a text editor theoretically empowered even non-technical people to be able to make the change in the case of emergency. In reality, a lack of documentation about how to edit the file meant that the people who could actually make a worthwhile change were limited to those who were intimately familiar with the raw code. We also had feature flags across many of our apps but failed to standardize them in any way. More standardization and documentation would have increased our emergency bus number significantly.

Possibly the most important takeaway from this whole experience is that we should have been doing these exercises more often, and that we needed to continue doing them. And so we did, until we were confident that all of our systems and applications would act as we expected, even in failure, for election day.

And when election day weekend happened and the scale jumped two to five times every day for four days, we knew not only how to handle any problems, but we knew that our applications and infrastructure would react predictably.

About the Author

Dylan Richard is the full-bearded technologist who, as Director for Engineering, quietly led the 40 engineers of Obama for America's Technology team to victory in 2012.

Previously, he helmed technical teams as Vice President of Engineering at skinnyCorp (home of threadless.com) and was a Senior Developer at Crate & Barrel.

Dylan lives in Chicago, Illinois, with his patient wife, Sarah, two amazing sons, a cat, and a rotating stable of old diesel VWs. His possible favorite claim to fame is having his chest-length beard proclaimed "fierce" by President Barack Obama.