acmqueue Case Study
Learning to Embrace Failure

A discussion with Jesse Robbins, Kripa Krishnan, John Allspaw, and Tom Limoncelli

It's very nearly the holiday shopping season and something is very wrong at a data center handling transactions for one of the largest online retail operations in the country. Some systems have failed, and no one knows why. Stress levels are off the charts while teams of engineers work around the clock for three days trying to recover.

The good news is that it's not a real disaster—though it could have been. Instead, it's an exercise designed to teach a company how to adapt to the inevitable: system failure. Things break; disaster happens; failure is real. Although no one—perhaps least of all software developers and systems engineers—likes to believe they can't prevent failure, the key to preparing for it is first to accept it.

Many operations are turning to resilience engineering not in hopes of becoming impervious to failure, but rather to become better able to adapt to it when it occurs. Resilience engineering is a familiar concept in high-risk industries such as aviation and health care, and now it's being adopted by large-scale Web operations as well.

In the early 2000s, Amazon created GameDay, a program designed to increase resilience by purposely injecting major failures into critical systems semi-regularly to discover flaws and subtle dependencies. Basically, a GameDay exercise tests a company's systems, software, and people in the course of preparing for a response to a disastrous event. Widespread acceptance of the GameDay concept has taken a few years, but many companies now see its value and have started to adopt their own versions. This discussion considers some of those experiences.

Participants include ***Jesse Robbins****, the architect of GameDay at Amazon, where he was officially called the Master of Disaster. Robbins used his training as a firefighter in developing GameDay, following similar principles of incident response. He left Amazon in 2006 and founded the Velocity Web Performance and Operations Conference, the annual O'Reilly meeting for people building at Internet scale. In 2008, he founded Opscode, which makes Chef, a popular framework for infrastructure automation. Running GameDay operations on a slightly smaller scale is* ***John Allspaw****, senior vice president of technological operations at Etsy. Allspaw's experience includes stints at Salon.com and Friendster before joining Flickr as engineering manager in 2005. He moved to Etsy in 2010. He also recently took over as chair of the Velocity conference from Robbins.*

Google's equivalent of GameDay is run by ***Kripa Krishnan****, who has been with the program almost from the time it started six years ago. She also works on other infrastructure projects, most of which are focused on the protection of users and their data.*

Moderating this discussion is a Google colleague, ***Tom Limoncelli****, who is a site reliability engineer. Well known in system administrator circles, he started out at Bell Labs and has written four books, most notably* Time Management for System Administrators *and* The Practice of System and Network Administration.

TOM LIMONCELLI Jesse, you've probably been involved in more GameDay exercises than anybody. What's the most important lesson you've taken away from that?

JESSE ROBBINS More than anything else, I've learned that the key to building resilient systems is accepting that failure happens. There's just no getting around it. That applies to the software discipline, as well as to the systems management and architectural disciplines. It also applies to managing people.

It's only after you've accepted the reality that failure is inevitable that you can begin the journey toward a truly resilient system. At the core of every resilience program—whether it's what you find at Google, Facebook, Etsy, Flickr, Yahoo, or Amazon—is the understanding that whenever you set out to engineer a system at Internet scale, the best you can hope for is to build a reliable software platform on top of components that are completely unreliable. That puts you in an environment where complex failures are both inevitable and unpredictable.

KRIPA KRISHNAN We've learned a few things as well. The most important of those lessons is that an untested disaster recovery plan isn't really a plan at all. We also know now that if doing something is hard, repetition is going to help make it easier. At Google scale, even if there's only a fraction of a one-percent chance of a failure occurring, that means it's a failure likely to occur multiple times. Our plan is to preemptively trigger the failure, observe it, fix it, and then repeat until that issue ceases to be one. In our most recent GameDay exercise, we retested some things that caused serious failures two to three years ago and were pleased to find they can now be resolved effortlessly.

We've also learned that real success doesn't come from just running a GameDay test once a year but instead from getting teams to test their services internally all year round. That said, GameDay gives us an opportunity to test some less-exercised links. For example, we design tests that require engineers from several groups who might not normally work together to interact with each other. That way, should a real large-scale disaster ever strike, these people will already have strong working relationships established.

Another point of emphasis is that none of the functions people are asked to perform on GameDay are significantly different from what they would do on any other day. That way, there won't be anything unusual to remember or panic about should a real failure occur.

TL Speaking of panic, what do these exercises look like from a software engineer's or system administrator's perspective? What do they actually go through during one of these exercises?

JR The program I designed at Amazon began with a series of companywide briefings advising everyone that we were about to do an exercise of a certain scale—say, something on the order of a full-scale data-center destruction. They didn't know which data center was going to be taken offline, and they didn't know exactly when it was going to happen, but they usually had a few months to make sure their systems were capable of surviving the sudden loss of a significant amount of capacity. The expectation was that they would use that time to make sure they had eliminated all the single points of failure they could find.

Once the event actually came to pass, people would typically receive a notification as part of our standard incident-management process, often including some information specific to their sites. For example, we might tell them that certain servers were no longer responding or that their service had lost some amount of capacity. Hopefully, their own standard monitoring tools would have picked all that up, but there have been instances where the GameDay exercise ended up taking out those monitoring capabilities. That isn't exactly what you would like to see, but still it's one of the classic defects these exercises are really good at exposing.

Sometimes you also expose what we call "latent defects"—problems that appear only because of the failure you've triggered. For example, you might discover that certain monitoring or management systems crucial to the recovery process end up getting turned off as part of the failure you've orchestrated. You would find some single points of failure you didn't know about that way. But, as Kripa said, as you do more and more of these exercises in a progressively intense and complicated way, it does start to become just a regular, ordinary part of doing business.

TL So as an engineer, I might be sitting at my desk when I get a page telling me that one of our data centers has just gone down. Would that be a page about something completely fictional, or are we talking about a situation where someone has actually disconnected a cable, so to speak?

JR In the exercises I designed, we used real failures. We would literally power off a facility—without notice—and then let the systems fail naturally and allowed the people to follow their processes wherever they led. In one of those exercises, we actually drew on my fire-service background to concoct a simulated fire. I wrote out the timing to the minute for that according to when we would expect certain things to happen as part of a full-scale fire response. Then, posing as some of the facilities guys, we called people in operations to update them on what was happening.

My view is that you want to make these drills seem as real as possible in order to expose those special systems and backdoor boxes that some system administrators have been holding onto in case of emergency. Not everything in these exercises can be simulated, and once you start powering down machines or breaking core software components, the problems that surface are real. Still, it's important to make it clear that the "disaster" at the core of the exercise is merely simulated so people on the periphery don't freak out. Otherwise, what happens in the course of that exercise ought to feel just as real as possible.

JOHN ALLSPAW Yes, and the exercise should also make people feel a little uncomfortable. The truth is that things often break in ways we can't possibly imagine. In the course of responding to those surprises, you get a chance to learn from your mistakes, of course. Ultimately, you also get an opportunity to gain confidence that the system you've built and the organization that's been built up around it are actually pretty resilient.

TL How does this work at Google, Kripa?

KK We normally give people three to four months' notice, telling them the event is set to take place within some particular week or month. The GameDay event itself is generally a round-the-clock, 72- to 96-hour exercise. Even though people are unaware of the exact timing, they know there's a period coming when there will be a lot of disruptions and they're going to be expected to respond to each of those disruptions as if it were a real event.

The disruptions we orchestrate range on the one hand from technical failures such as powering down a full data center or purposely corrupting data on our back ends to exercises that test the human element—for example, creating a situation where an entire team is rendered incommunicado for 72 hours and other teams are forced to work around them to simulate what might happen following an earthquake. Tests involving less-severe issues might last only a few hours, whereas the larger-scale tests tend to run a few days. The idea is always to discover as much as we can about how the company performs under stress with reduced capacity over an extended period of time.

Just to give you a sense of scale, we typically have hundreds of engineers working around the clock for the duration of one of these exercises. We usually have one set of engineers who respond to the test and another set who act as "proctors" monitoring the test. The proctors keep a close eye on the communications flying back and forth across IRC (Internet relay chat) channels. We also staff "war rooms" to make sure things don't get so out of control that they end up making the situation worse or impacting the production environment.

Roughly speaking, during the first 24 hours of the test, it's all about the initial response. The big problems generally surface then. Between 24 and 48 hours, a lot of routine team-to-team testing takes place. Independent engineering teams write tests for their counterparts in other locations, and they end up doing a lot of bidirectional testing. Then, by the 72-hour mark, signs of exhaustion really start to show. And it turns out exhaustion and other human factors are an important part of what we test. That's because, in a real emergency, you might not have the option of handing off work at the end of your shift.

You asked earlier about whether the paging that gets sent out is fictional or real. We actually do page people for real, but we also try to make it clear the page relates to a GameDay test. The last thing we want is to give a test priority over a real production issue. Still, we've had situations where we learned about the "latent defects" Jesse was talking about, such as when we discovered the paging infrastructure was located in a facility we'd brought down, which ended up causing a pretty serious ruckus.

TL I think everybody goes through that. You know you're conducting a successful exercise when the first thing you learn is that some critical part of your response framework is in scope for the test.

KK Yes, but the test is really successful only if, when you repeat it later, you're able to ensure that everything works fine. In this particular case, the offending test caused tons of pages to queue up in the infrastructure, with none of them getting out to the right people. Then, when we finally managed to undo the test, a lot of pagers got blasted with thousands of pages—with no way to figure out which ones were real and which ones were GameDay-related. So you *know* that got fixed!

Introducing GameDay scenarios into some of these Web-scale companies has initiated a difficult cultural shift from a steadfast belief that systems should never fail—and if they do, focusing on who's to blame—to actually forcing systems to fail. Rather than expending resources on building systems that don't fail, the emphasis has started to shift to how to deal with systems swiftly and expertly once they do fail—because fail they will.

Failure can be a hard sell, but converts may indeed be won over gradually when they see how much they can learn from GameDay exercises. Much of the value in running such an exercise comes from changing the collective mindset of the engineers who designed and built the systems. It's not easy for them to watch as their systems fail and to see what consequences come of that. Over time, however, they start to gain confidence that the systems and practices they rely upon are actually pretty resilient.

Companies that buy into the GameDay philosophy also hope to invoke a more just culture—one in which people can be held accountable without being blamed, or punished, for failure.

TL You have managed to establish yourself in the business of orchestrating disasters, but traditionally management has considered any and all outages to be completely unacceptable. Does this suggest a major cultural shift?

KK We definitely went through a major cultural shift over the first couple of years of our disaster-recovery testing program. What really worked in our favor, though, was that we had a solid sponsor—our VP, Ben Treynor—who strongly believed that the only way you can be sure something works as expected is to test it. But even with strong support like that, we still got some significant pushback. Some nonengineering departments in particular saw only the risks and the investments that would be involved, and some operations teams just couldn't see the advantage of GameDay-style testing over the continuous testing programs they already had under way. In fact, the most important predictor of an organization's willingness to cooperate proved to be how the people in that organization had handled previous failures. If those earlier outages had resulted in engineering investigations that didn't affix blame to individuals but instead just looked for root causes, the organization almost invariably proved eager to participate. On the other hand, if those earlier outages had led to hunts for the guilty, then the organization generally proved to be more reluctant.

Today, however, GameDay is embraced across the board. Our most recent exercise involved 20 times as many teams writing tests as we had five years ago, with participation coming from both technical and business groups. Now, when we find things that are broken, no one feels ashamed. They just accept learning about the problem as an opportunity to go back and fix it. Which is to say that everyone now seems to grasp that the whole point of the exercise is to find issues so remedies or course corrections can be implemented proactively.

JR I had a somewhat different experience at Amazon back in 2003/2004 when the idea of horizontal scalability across unreliable hardware wasn't yet a fully formed concept—in fact, we were pretty much out at the hairy edge and feeling our way. We did recognize, however, that the scale and complexity of our failures were growing as our sites grew in size and number. That's when it dawned on us that there were probably only two approaches we could take: one would be to spend as much money as necessary to make things more reliable; and the other was that, as an organization, we could choose to embrace the idea that failure happens.

I enjoyed great executive support early on since it was fairly clear we weren't going to solve this challenge just by throwing money at it. So right away our executive team bought into the notion that triggering failures in a controlled manner represented an opportunity for us to learn some really important big lessons at a much lower cost and with far less disruption than would be the case if we just waited for problems to surface on their own.

That became the fundamental basis for running a GameDay-type exercise. That is, you can't choose whether or not you're going to have failures—they are going to happen no matter what—but you can choose in many cases *when* you're going to learn the lessons. The thing is, you really want to be able to expose the defects in your systems on your terms and at the time of your choosing, and there's just no other way to find some of those problems than to trigger failures.

Once an organization buys into that thinking, the culture tends to change pretty quickly. People who go through the process, difficult as it might be, find something valuable, and many of them soon become powerful advocates. It generally doesn't take long before you've got a fair number of people who accept that failure happens and so become frankly quite willing to have the fun of breaking stuff in order to expose the problems they know are lurking in there somewhere. Out of that comes a new operational culture, but it's one that can be built only through a series of exercises.

You can't do just one of these exercises and then be done with it. This is one of the great lessons we took from the fire service, where it's understood you have to keep training and drilling regularly to develop the operating competencies that can only come over time.

TL I just have to point out that you said something quite extraordinary when you mentioned how quickly the executives at Amazon embraced the notion that failure is unavoidable. Traditionally executives have been taught that any time a major system failure happens, the first thing they should do is to fire some people just to show everyone they're in control of the situation.

JR We probably had a particular advantage at Amazon in that much of the organization already was experienced with process optimization around fulfillment centers. Basically, we already had something of a plant-operations mentality.

It also helped that around that same time we had a few big outages. As a result, people were especially open and receptive to doing something different. At companies where there hasn't been an outage for a while, it might prove a little harder to sell the GameDay concept because, you know, complacency builds. What I always recommend to people at companies that haven't accepted the wisdom of embracing failure is that they should keep an eye out for any failures that come up, along with any other issues that might give them the ammunition they need to argue for a small GameDay experiment. Of course, then, should the experiment prove to be at all successful, they ought to publicize the heck out of it by pointing out the problems they managed to address in a controlled manner instead of just waiting around for disaster to strike.

JA There are some lessons in what you've just said for any organization that aims to be resilient—particularly those that rely on complex systems. And by *resilient*, I mean the ability to sustain operations before, during, and after an unexpected disturbance—something that's outside the bounds of what the designer had anticipated.

Erik Hollnagel, a pioneer in the area of resilience engineering, has noted that the four cornerstones of resilience are: (1) know what to expect (anticipation); (2) know what to look for (monitoring); (3) know what to do (how to respond); and (4) know what just happened (learning). That last one is where things like postmortems come into play.

The view Hollnagel articulates is a little different from the traditional one, which holds that screw-ups and outages result from mistakes made by certain individuals. When you're talking about a complex product, however, it's likely that any flaws actually derive from some failure in the overall design process or something that went wrong in one of the many different product-development steps along the way. Which is to say the responsibility for any problems that surface later really should be borne collectively. So when all the blame ends up being put on an individual—most likely an engineer—well, that's just ridiculously reductionist. It's like saying, "Here's the guy. He made the mistake, and everything came crumbling down because of that. So what's our problem here? *He's* our problem." That's nothing more than a classic illustration of hindsight bias.

The trick, of course, is to get people throughout the organization to start building their anticipation muscles by thinking about what might possibly go wrong. You already see an aspect of this in the current craze for continuous deployment, since that focuses on how to keep systems up while failed components are swapped out for repairs or replacement. If one part of your Web site is broken, it shouldn't bring down the whole site, right? They don't shut down the Brooklyn Bridge just because one lane is kaput.

TL That covers the anticipation aspect. What about some of the other cornerstones of resilience?

JA One of the most important aspects—and one that gets far too little attention—is learning from failure. The traditional response has been: guy makes a mistake… site goes down… have a meeting… fire that dude. He must have been careless or negligent, or maybe there was even a willful violation of procedures.

But I'd like to believe our professional culture is going to come around to embracing the wisdom in holding people accountable without necessarily blaming them. The medical profession and the aviation industry have both benefited from that attitude. Beyond the obvious issues of basic fairness, believing that "human error" is the single root cause of some problem is just pure folly. First of all, when you're dealing with something as complex and inherently flawed as the Web infrastructure, the very notion of having only a single root cause is laughable. Then to pass the problem off as something that might be fixed by firing someone means you really haven't learned anything—and that's sure not going to help you become more resilient in the future.

The only way you can learn something useful from a failure is to find out all you can about the actions that led up to the outage—and that means finding out why it seemed to make sense to take those particular steps at that particular time. There are some human-factors tests that can help with this. One that we use at Etsy is called the substitution test, and it comes in handy in figuring out why somebody decided to run a command that ended up bringing down the site. We'll grab another engineer who had no involvement in the problem situation, and we'll fill him in on the context and all the particulars known to the operator at the time. Then we'll ask that engineer, "What would you have done in that situation?" Almost every single time, he will tell us he would have run exactly the same command.

The problem therefore isn't due to a lack of training, a lack of intelligence, or any other personal failing. The solution isn't just to fire the guy. In fact, you want to keep that guy and drill down into what led to the mistake. Find out why he thought it made sense to do what he did.

I've already announced publicly on several occasions that I'm never going to fire anybody for taking down a site I'm responsible for. I don't think a lot of other engineering leaders are willing to go that far. In fact, I think it makes a couple of people in my own organization pretty nervous.

JR Yes, but in many cases, you can only learn from those mistakes.

JA The flipside to this is that some see resilience engineering as being not only about looking at the future and trying to anticipate failures, but also about deconstructing scenarios and then putting the pieces back together again to better understand how we've responded in the past so we can think about how we might respond better in the future. That's just another way of bolstering our powers of anticipation.

The really mind-blowing thing whenever you do that is to consider all the times when your site *didn't* fail, even though it probably should have. In addition to looking hard at why you ended up suffering some particular failure, you might want to turn that around and ask why you aren't having failures all the time.

GameDay exercises start out by breaking things in ways that can be easily imagined—in fact, in ways that are typically scripted—but then things often end up going off that script in some rather significant and unexpected ways. This adds a level of complexity that people must learn to manage. Why did these things happen and how should they be dealt with? The volume of information collected from a GameDay exercise can be staggering. Then it becomes important to come up with ways to process all that information and distribute it to the appropriate people so it can be put to use when needed.

Human factors are also a complication in exercises of this magnitude. GameDays can go on for a few days, so keeping the participants focused can be a challenge. You don't want people to become complacent, so *there's a lot of value in keeping the pressure on, to a reasonable degree. Yet people make mistakes in stressful circumstances, so you have to compensate for the exhaustion factor by rotating people in at appropriate intervals. Often people don't want to give up until they have solved the problem, though their decision-making abilities may have become degraded by that point.*

TL You've all said that flushing out latent defects is one of the primary motivations for triggering failures. Is there ever a time when you can declare victory?

KK Not really. One reason is that for organizations at our scale, systems are constantly evolving as new layers are added, and just taking inventory of all those services alone can require considerable effort. Complexities are introduced as new capabilities are developed. And even more complexity is introduced whenever acquisitions require you to merge in new code bases—which, of course, only makes it harder to anticipate where something might break. It gets progressively harder to see where our dependencies are and what might lead to cascading failures. The only way to find those latent defects is to run exercises where we can trigger actual failures.

A few examples might be the best way to illustrate this. We've had situations where we brought down a network in, say, São Paulo, only to find that in doing so we broke our links in Mexico. That seems totally bizarre on the face of it, but as you dig down, you end up finding some dependency no one knew about previously. And we still wouldn't know about it to this day if we hadn't caused that network to fail.

We also had a case once where we turned off a data center only to find a good percentage of our machines there wouldn't come back online even when we tried to power-cycle them. A full night and a day later, we figured out that we had run out of DHCP (Dynamic Host Configuration Protocol) leases. Testing tends to surface silly stuff like that as well, but even silly stuff can have dire consequences.

JR For the most part, we've been talking about a class of failures you learn about only in the course of one of these outages, but we also see problems and failures crop up as we're working to recover from the outages. Since leaving Amazon, I've helped a number of other organizations and have run into a number of recovery issues related to scale and complexity. As organizations grow, the tools they have for configuration management generally don't keep up. They find themselves in situations where they need to deploy code to 1,000 new boxes, but that turns out to be something their software deployment system hasn't been sized for.

TL How do you manage to process all that you've learned over the course of one of these exercises?

KK Before, during, and after the test, we repeatedly emphasize to people that they should file bug reports about every single broken thing they've found. That applies to people processes, as well as to bugs in the code. Those of us who manage the exercise also file our own set of bug reports. We have a rotating team of about 50 people in war rooms around the world—predominantly volunteers—each focusing on different parts of the operation. We're all constantly taking down notes on Post-its and whiteboards as we find things that are broken.

Following the exercise, we spend a couple of weeks assimilating all that information into a report and sorting out a punch list of fixes. This is possibly the hardest part of the job. Hundreds of services generate a thousand issues. This gets processed and, more importantly, prioritized. Complex companywide issues need buy-in from several organizations, whereas service-specific bugs just need to be filed against each team. Really, that's all there is to it. Once the right information has been put in front of the right people, we can—with trivial effort on our part—close out 80 to 85 percent of the issues raised by each GameDay exercise.

No matter how those changes end up being made, the most important point is for organizations to realize how important it is to constantly test their systems and the integration points between different teams, as well as between all the underlying technology. That's particularly important for Internet-based companies, since the infrastructure and the software they rely upon are continuously changing. There's just no way of knowing what your altered system is going to be able to sustain without putting it to the test, and that means triggering actual failures through GameDay-style exercises.

JA It's also helpful for all of us to remember that the field of Web engineering is only about 11 years old. Meanwhile, some fields of engineering have many decades of experience in building and maintaining complex systems, so there's something to be gained from considering some of those other domains to see what we might learn in terms of design principles and best practices that can be adapted for our own purposes. GameDay exercises demonstrate that we're starting to do just that.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org