19

ENABLE AND INJECT LEARNING INTO DAILY WORK

When we work within a complex system, it is impossible for us to predict all the outcomes for the actions we take. This contributes to unexpected and sometimes catastrophic accidents, even when we use static precautionary tools, such as checklists and runbooks, which codify our current understanding of the system.

To enable us to safely work within complex systems, our organizations must become ever-better at self-diagnostics and self-improvement and must be skilled at detecting problems, solving them, and multiplying the effects by making the solutions available throughout the organization. This creates a dynamic system of learning that allows us to understand our mistakes and translate that understanding into actions that helps to prevent those mistakes from recurring in the future.

The result is what Dr. Steven Spear describes as resilient organizations who are "skilled at detecting problems, solving them, and multiplying the effect by making the solutions available throughout the organization."[1] These organizations can heal themselves. "For such an organization, responding to crises is not idiosyncratic work. It is something that is done all the time. It is this responsiveness that is their source of reliability."[2]

AWS US-East and Netflix (2011)

A striking example of the incredible resilience that can result from these principles and practices was seen on April 21, 2011, when an entire availability zone in Amazon Web Services' (AWS) US-East region went down, taking down virtually all of the organizations who depended on it, including Reddit and Quora.[3*] However, Netflix was a surprising exception, seemingly unaffected by this massive AWS outage.

Following the event, there was considerable speculation about how Netflix kept their services running. A popular theory ran that since Netflix was one of the largest customers of Amazon Web Services, it was given some special treatment that allowed them to keep running. However, a *Netflix Engineering* blog post explained that their architectural design decisions in 2009 enabled their exceptional resilience.

Back in 2008, Netflix's online video delivery service ran on a monolithic J2EE application hosted in one of their data centers. However, starting in 2009, they began re-architecting this system to be what they called *cloud native*—it was designed to run entirely in the Amazon public cloud and to be resilient enough to survive significant failures.[5]

One of their specific design objectives was to ensure Netflix services kept running, even if an entire AWS availability zone went down, such as what happened with US-East. To do this required that their system be loosely coupled, with each component having aggressive timeouts and circuit breakers[†] to ensure that failing components didn't bring the entire system down.

Instead, each feature and component was designed to gracefully degrade. For example, during traffic surges that created CPU-usage spikes, instead of showing a list of movies personalized to the user, they would show static content, such as cached or un-personalized results, which required less computation.[6]

Furthermore, as the blog post explained, in addition to implementing these architectural patterns, they also built and had been running a surprising and audacious service called *Chaos Monkey*, which simulated AWS failures by constantly and randomly killing production servers. They did so because they wanted all "engineering teams to be used to a constant level of failure in the cloud" so that services could "automatically recover without any manual intervention."[7]

In other words, the Netflix team ran Chaos Monkey to gain assurance that they had achieved their operational resilience objectives, constantly injecting failures into their pre-production and production environments.

As one might expect, when they first ran Chaos Monkey in their production environments, services failed in ways they never could have predicted or imagined. By constantly finding and fixing these issues during normal working hours, Netflix engineers quickly and iteratively created a more resilient service, while simultaneously creating organizational learnings (during normal working hours!) that enabled them to evolve their systems far beyond their competition.

Chaos Monkey is just one example of how learning can be integrated into daily work. The story also shows how learning organizations think about failures, accidents, and mistakes—as an opportunity for learning and not something to be punished. This chapter explores how to create a system of learning and how to establish a *just culture*, as well as how to routinely rehearse and deliberately create failures to accelerate learning.

Establish a Just, Learning Culture

One of the prerequisites for a learning culture is that when accidents occur (which they undoubtedly will), the response to those accidents is seen as "just." Dr. Sidney Dekker, who helped codify some of the key elements of safety culture and coined the term *just culture*, writes, "When responses to incidents and accidents are seen as unjust, it can impede safety investigations, promoting fear rather than mindfulness in people who do safety-critical work, making organizations more bureaucratic rather than more careful, and cultivating professional secrecy, evasion, and self-protection."[8]

This notion of punishment is present, either subtly or prominently, in the way many managers have operated during the last century. The thinking goes, in order to achieve the goals of the organization, leaders must command, control, establish procedures to eliminate errors, and enforce compliance of those procedures.

Dr. Dekker calls this notion of eliminating error by eliminating the people who caused the errors the *bad apple theory*. He asserts that this is invalid, because "human error is not our cause of troubles; instead, human error is a consequence of the design of the tools that we gave them."[9]

If accidents are not caused by "bad apples" but rather are due to inevitable design problems in the complex system that we created, then instead of "naming, blaming, and shaming" the person who caused the failure, our goal should always be to maximize opportunities for organizational learning, continually reinforcing that we value actions that expose and share more widely the problems in our daily work. This is what enables us to improve the quality and safety of the system we operate within and reinforce the relationships between everyone who operates within that system.

By turning information into knowledge and building the results of the learning into our systems, we start to achieve the goals of a just culture, balancing the needs for safety and accountability. As John Allspaw, CTO of Etsy, states, "Our goal at Etsy is to view mistakes, errors, slips, lapses, and so forth with a perspective of learning."[10]

When engineers make mistakes and feel safe when giving details about it, they are not only willing to be held accountable, but also enthusiastic in helping the rest of the company avoid the same error in the future. This is what creates organizational learning. On the other hand, if we punish that engineer, everyone is dis-incentivized to provide the necessary details to get an understanding of the mechanism, pathology, and operation of the failure, which guarantees that the failure will occur again.

Two effective practices that help create a just, learning-based culture are blameless post-mortems (also called retrospectives or learning reviews) and the controlled introduction of failures into production to create opportunities to practice for the inevitable problems that arise within complex systems. We will first look at a retrospective and follow that with an exploration of why failure can be a good thing.

Schedule Retrospective Meetings after Accidents Occur

To help enable a just culture, when accidents and significant incidents occur (e.g., failed deployment, production issue that affected customers), we should conduct a retrospective after the incident has been resolved.[‡] Retrospectives help us examine "mistakes in a way that focuses on the situational aspects of a failure's mechanism and the decision-making process of individuals proximate to the failure."[11]

To do this, we conduct the retrospective as soon as possible after the accident occurs and before memories and the links between cause and effect fade or circumstances change. (Of course, we wait until after the problem has been resolved so as not to distract the people who are still actively working on the issue.)

In the meeting, we will do the following:

- Construct a timeline and gather details from multiple perspectives on failures, ensuring we don't punish people for making mistakes.
- Empower all engineers to improve safety by allowing them to give detailed accounts of their contributions to failures.
- Enable and encourage people who do make mistakes to be the experts who educate the rest of the organization on how not to make them in the future.
- Accept that there is always a discretionary space where humans can decide to take action or not, and that the judgment of those decisions lies in hindsight.
- Propose countermeasures to prevent a similar accident from happening in the future and ensure these countermeasures are recorded with a target date and an owner for follow-up.

To enable us to gain this understanding, the following stakeholders need to be present at the meeting:

- the people involved in decisions that may have contributed to the problem
- the people who identified the problem
- the people who responded to the problem
- the people who diagnosed the problem
- the people who were affected by the problem
- anyone else who is interested in attending the meeting

Our first task in the retrospective is to record our best understanding of the timeline of relevant events as they occurred. This includes all actions we took and at what time (ideally supported by chat logs, such as IRC or Slack), what effects we observed (ideally in the form of the specific metrics from our production telemetry, as opposed to merely subjective narratives), all investigation paths we followed, and what resolutions were considered.

To enable these outcomes, we must be rigorous about recording details and reinforcing a culture within which information can be shared without fear of punishment or retribution. Because of this, especially for our first few retrospectives, it may be helpful to have the meeting led by a trained facilitator who wasn't involved in the accident.

During the meeting and the subsequent resolution, we should explicitly disallow the phrases "would have" or "could have," as they are *counterfactual* statements that result from our human tendency to create possible alternatives to events that have already occurred.

Counterfactual statements, such as "I could have . . . " or "If I had known about that, I should have . . . ," frame the problem in terms of the *system as imagined* instead of in terms of the *system that actually exists*, which is the context we need to restrict ourselves to. (See Appendix 8.)

One of the potentially surprising outcomes of these meetings is that people will often blame themselves for things outside of their control or question their own abilities. Ian Malpass, an engineer at Etsy observes,

> In that moment when we do something that causes the entire site to go down, we get this "ice-water down the spine" feeling, and likely the first thought through our head is, "I suck and I have no idea what I'm doing." We need to stop ourselves from doing that, as it is route to madness, despair, and feelings of being an imposter, which is something that we can't let happen to good engineers. The better question to focus on is, "Why did it make sense to me when I took that action?"[12]

In the meeting, we must reserve enough time for brainstorming and deciding which countermeasures to implement. Once the countermeasures have been identified, they must be prioritized and given an owner and timeline for implementation. Doing this further demonstrates that we value improvement of our daily work more than daily work itself.

Dan Milstein, one of the principal engineers at Hubspot, writes that he begins all retrospectives by saying, “We’re trying to prepare for a future where we’re as stupid as we are today.”[13] In other words, it is not acceptable to have a countermeasure to merely “be more careful” or “be less stupid”—instead, we must design real countermeasures to prevent these errors from happening again.

Examples of such countermeasures include new automated tests to detect dangerous conditions in our deployment pipeline, adding further production telemetry, identifying categories of changes that require additional peer review, and conducting rehearsals of this category of failure as part of regularly scheduled game day exercises.

Publish Our Retrospective Reviews as Widely as Possible

After we conduct a retrospective, we should widely announce the availability of the meeting notes and any associated artifacts (e.g., timelines, IRC chat logs, external communications). This information should (ideally) be placed in a centralized location where our entire organization can access it and learn from the incident. Conducting retrospectives is so important that we may even prohibit production incidents from being closed until the retrospective has been completed.

Doing this helps us translate local learnings and improvements into global learnings and improvements. Randy Shoup, former engineering director for Google App Engine, describes how documentation of retrospectives can have tremendous value to others in the organization: “As you can imagine at Google, everything is searchable. All the retrospective documents are in places where other Googlers can see them. And trust me, when any group has an incident that sounds similar to something that happened before, these retrospective documents are among the first documents being read and studied.”[14§]

Widely publishing retrospectives and encouraging others in the organization to read them increases organizational learning. It is also becoming increasingly commonplace for online service companies to publish retrospectives for customer-impacting outages. This often significantly increases the transparency we have with our internal and external customers, which in turn increases their trust in us.

This desire to conduct as many retrospective meetings as necessary at Etsy led to some problems—over the course of four years, Etsy accumulated a large number of post-mortem meeting notes (retrospectives) in wiki pages, which became increasingly difficult to search, save, and collaborate from. To help with this issue, they developed a tool called Morgue to easily record aspects of each accident, such as the incident MTTR and severity, better address time zones (which became relevant as more Etsy employees were working remotely), and other data, such as rich text in Markdown format, embedded images, tags, and history.[16]

Morgue was designed to make it easy for the team to record:

- whether the problem was due to a scheduled or unscheduled incident
- the retrospective owner
- relevant IRC chat logs (especially important for 3 AM issues when accurate note-taking may not happen)
- relevant JIRA tickets for corrective actions and their due dates (information particularly important to management)
- links to customer forum posts (where customers complain about issues)

After developing and using Morgue, the number of recorded retrospectives at Etsy increased significantly compared to when they used wiki pages, especially for P2, P3, and P4 incidents (i.e., lower severity problems).[17] This result reinforced the hypothesis that if they made it easier to document retrospectives through tools such as Morgue, more people would record and detail the outcomes of their retrospective, enabling more organizational learning.

CONTINUOUS LEARNING

Conducting retrospectives does more than help us learn from failure. DORA's *2018 State of DevOps Report* found that they contribute to our culture, helping teams feel better about sharing information, taking smart risks, and understanding the value of learning. In addition, the research found that elite performers were 1.5 times more likely to consistently hold retrospectives and use them to improve their work—so these elite teams continue to reap the benefits.[18]

Dr. Amy C. Edmondson, Novartis Professor of Leadership and Management at Harvard Business School and co-author of *Building the Future: Big Teaming for Audacious Innovation*, writes:

> Again, the remedy—which does not necessarily involve much time and expense—is to reduce the stigma of failure. Eli Lilly has done this since the early 1990s by holding "failure parties" to honor intelligent, high-quality scientific experiments that fail to achieve the desired results. The parties don't cost much, and redeploying valuable resources—particularly scientists—to new projects earlier rather than later can save hundreds of thousands of dollars, not to mention kickstart potential new discoveries.[19]

Decrease Incident Tolerances to Find Ever-Weaker Failure Signals

Inevitably, as organizations learn how to see and solve problems efficiently, they need to decrease the threshold of what constitutes a problem in order to keep learning. To do this, we seek to amplify weak failure signals. For example, as described in Chapter 4, when Alcoa was able to reduce the rate of workplace accidents so that they were no longer commonplace, Paul O'Neill, CEO of Alcoa, started to be notified of accident near-misses in addition to actual workplace accidents.[20]

Dr. Spear summarizes O'Neill's accomplishments at Alcoa when he writes, "Though it started by focusing on problems related to workplace safety, it soon found that safety problems reflected process ignorance and that this ignorance would also manifest itself in other problems such as quality, timeliness, and yield versus scrap."[21]

When we work within complex systems, this need to amplify weak failure signals is critical to averting catastrophic failures. The way NASA handled failure signals during the space shuttle era serves as an illustrative example. In 2003, sixteen days into the *Columbia* space shuttle mission, it exploded as it re-entered the earth's atmosphere. We now know that a piece of insulating foam had broken off the external fuel tank during takeoff.

However, prior to *Columbia's* re-entry, a handful of mid-level NASA engineers had reported this incident, but their voices had gone unheard. They observed the foam strike on video monitors during a post-launch review session and immediately notified NASA's managers, but they were told that the foam issue was nothing new. Foam dislodgement had damaged shuttles in previous launches but had never resulted in an accident. It was considered a maintenance problem and not acted upon until it was too late.[22]

Michael Roberto, Richard M. J. Bohmer, and Amy C. Edmondson wrote in a 2006 article for *Harvard Business Review* how NASA culture contributed to this problem. They described how organizations are typically structured in one of two models: a *standardized model*, where routine and systems govern everything, including strict compliance with timelines and budgets, or an *experimental model*, where every day, every exercise and every piece of new information is evaluated and debated in a culture that resembles a research and design (R&D) laboratory.[23]

They observe, "Firms get into trouble when they apply the wrong mind-set to an organization [which dictates how they respond to *ambiguous threats* or, in the terminology of this book, *weak failure signals*]. . . . By the 1970s, NASA had created a culture of rigid standardization, promoting to Congress the space shuttle as a cheap and reusable spacecraft."[24]

NASA favored strict process compliance instead of an experimental model where every piece of information needed to be evaluated as it occured without bias. The absence of continuous learning and experimentation had dire consequences. The authors conclude that it is culture and mindset that matters, not just "being careful"—as they write, "vigilance alone will not prevent ambiguous threats [weak failure signals] from turning into costly (and sometimes tragic) failures."[25]

Our work in the technology value stream, like space travel, should be approached as a fundamentally experimental endeavor and managed that way. All work we do is a potentially important hypothesis and a source of data, rather than a routine application and validation of past practice. Instead of treating technology work as entirely standardized, where we strive for process compliance, we must continually seek to find ever-weaker failure signals so that we can better understand and manage the system we operate in.

Redefine Failure and Encourage Calculated Risk-Taking

Leaders of an organization, whether deliberately or inadvertently, reinforce the organizational culture and values through their actions. Audit, accounting, and ethics experts have long observed that the "tone at the top" predicts the likelihood of fraud and other unethical practices. To reinforce our culture of learning and calculated risk-taking, we need leaders to continually reinforce that everyone should feel both comfortable with and responsible for surfacing and learning from failures.

On failures, Roy Rapoport from Netflix observes,

> What the *2014 State of DevOps Report* proved to me is that high performing DevOps organizations will fail and make mistakes more often. Not only is this okay, it's what organizations need! You can even see it in the data: if high performers are performing thirty times more frequently but with only half the change failure rate, they're obviously having more failures. . . .[26]
>
> I was talking with a co-worker about a massive outage we just had at Netflix—it was caused by, frankly, a dumb mistake. In fact, it was

> caused by an engineer who had taken down Netflix twice in the last eighteen months. But, of course, this is a person we'd never fire. In that same eighteen months, this engineer moved the state of our operations and automation forward not by miles but by light-years. That work has enabled us to do deployments safely on a daily basis, and has personally performed huge numbers of production deployments.[27]

He concludes, "DevOps must allow this sort of innovation and the resulting risks of people making mistakes. Yes, you'll have more failures in production. But that's a good thing and should not be punished."[28]

Inject Production Failures to Enable Resilience and Learning

As we saw in the chapter introduction, injecting faults into the production environment (such as using Chaos Monkey) is one way we can increase our resilience. In this section, we describe the processes involved in rehearsing and injecting failures into our system to confirm that we have designed and architected our systems properly, so that failures happen in specific and controlled ways. We do this by regularly (or even continuously) performing tests to make certain that our systems fail gracefully.

As Michael Nygard, author of *Release It! Design and Deploy Production-Ready Software*, comments, "Like building crumple zones into cars to absorb impacts and keep passengers safe, you can decide what features of the system are indispensable and build in failure modes that keep cracks away from those features. If you do not design your failure modes, then you will get whatever unpredictable—and usually dangerous—ones happen to emerge."[29]

Resilience requires that we first define our failure modes and then perform testing to ensure that these failure modes operate as designed. One way we do this is by injecting faults into our production environment and rehearsing large-scale failures so we are confident we can recover from accidents when they occur, ideally without even impacting our customers.

The 2012 story about Netflix and the Amazon AWS US-East outage presented at the beginning of this chapter is just one example. An even more interesting example of resilience at Netflix was during the "Great Amazon Reboot of 2014," when nearly 10% of the entire Amazon EC2 server fleet had to be rebooted to apply an emergency Xen security patch.[30]

As Christos Kalantzis of Netflix Cloud Database Engineering recalled, "When we got the news about the emergency EC2 reboots, our jaws dropped. When we got the list of how many Cassandra nodes would be affected, I felt ill."[31] But, Kalantzis continues, "Then I remembered all the Chaos Monkey exercises we've gone through. My reaction was, 'Bring it on!'"[32]

Once again, the outcomes were astonishing. Of the 2,700+ Cassandra nodes used in production, 218 were rebooted, and twenty-two didn't reboot successfully. As Kalantzis and Bruce Wong from Netflix Chaos Engineering wrote, "Netflix experienced 0 downtime that weekend. Repeatedly and regularly exercising failure, even in the persistence [database] layer, should be part of every company's resilience planning. If it wasn't for Cassandra's participation in Chaos Monkey, this story would have ended much differently."[33]

Even more surprising, not only was no one at Netflix working active incidents due to failed Cassandra nodes, no one was even in the office—they were in Hollywood at a party celebrating an acquisition milestone.[34] This is another example demonstrating that proactively focusing on resilience often means that a firm can handle events that may cause crises for most organizations in a manner that is routine and mundane.[¶] (See Appendix 9.)

Institute Game Days to Rehearse Failures

In this section, we describe specific disaster recovery rehearsals called game days, a term popularized by Jesse Robbins, one of the founders of the Velocity Conference community and co-founder of Chef. While at Amazon, Robbins was responsible for creating programs to ensure site availability and was widely known internally as the "Master of Disaster."[36] The concept of game days comes from the discipline of resilience engineering. Robbins defines *resilience engineering* as "an exercise designed to increase resilience through large-scale fault injection across critical systems."[37]

Robbins observes that "whenever you set out to engineer a system at 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."[38]

Consequently, we must ensure that services continue to operate when failures occur, potentially throughout our system, ideally without crisis or even manual intervention. As Robbins quips, "a service is not really tested until we break it in production."[39]

Our goal for game day is to help teams simulate and rehearse accidents to give them the ability to practice. First, we schedule a catastrophic event, such as the simulated destruction of an entire data center, to happen at some point in the future. We then give teams time to prepare, to eliminate all the single points of failure, and to create the necessary monitoring procedures, failover procedures, etc.

Our game day team defines and executes drills, such as conducting database failovers (i.e., simulating a database failure and ensuring that the secondary database works) or turning off an important network connection to expose problems in the defined processes. Any problems or difficulties that are encountered are identified, addressed, and tested again.

At the scheduled time, we then execute the outage. As Robbins describes, at Amazon they "would literally power off a facility—without notice—and then let the systems fail naturally and [allow] the people to follow their processes wherever they led."[40]

By doing this, we start to expose the *latent defects* in our system, which are the problems that appear only because of having injected faults into the system. Robbins explains, "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."[41] These exercises are then conducted in an increasingly intense and complex way with the goal of making them feel like just another part of an average day.

By executing game days, we progressively create a more resilient service and a higher degree of assurance that we can resume operations when inopportune events occur, as well as create more learnings and a more resilient organization.

An excellent example of simulating disaster is Google's Disaster Recovery Program (DiRT). Kripa Krishnan, Technical Program Director at Google at the time, had led the program for over seven years. During that time, they had simulated an earthquake in Silicon Valley, which resulted in the entire Mountain View campus being disconnected from Google; major data centers having complete loss of power; and even aliens attacking cities where engineers resided.[42]

As Krishnan wrote, "An often-overlooked area of testing is business process and communications. Systems and processes are highly intertwined, and separating testing of systems from testing of business processes isn't realistic: a failure of a business system will affect the business process, and conversely a working system is not very useful without the right personnel."[43]

Some of the learnings gained during these disasters included:[44]

- When connectivity was lost, the failover to the engineer workstations didn't work.
- Engineers didn't know how to access a conference call bridge or the bridge only had capacity for fifty people or they needed a new conference call provider who would allow them to kick off engineers who had subjected the entire conference to hold music.
- When the data centers ran out of diesel for the backup generators, no one knew the procedures for making emergency purchases through the supplier, resulting in someone using a personal credit card to purchase $50,000 worth of diesel.

By creating failure in a controlled situation, we can practice and create the playbooks we need. One of the other outputs of game days is that people actually know who to call and know who to talk to—by doing this, they develop relationships with people in other departments so they can work together during an incident, turning conscious actions into unconscious actions that are able to become routine.

➡ CASE STUDY: NEW TO SECOND EDITION

Turning an Outage into a Powerful Learning Opportunity at CSG (2021)

CSG is North America's largest SaaS-based customer care and billing provider, with over sixty-five million subscribers and a tech stack that covers everything from Java to mainframe. At the DevOps Enterprise Summit in 2020, Erica Morrison, Vice President of Software Engineering, shared the story of CSG's worst outage—the result of a complex system failure that pushed CSG beyond the limits of its response systems, processes, and culture.[45]

But in the face of that adversity, they were able to find opportunity and use the lessons they learned to improve how they understand incidents, respond to them, and prevent them in the first place.

The 2/4 Outage, as it later came to be known, lasted thirteen hours. It started abruptly and large portions of CSG's product were unavailable. On the initial calls as the outage began, the team was troubleshooting blind as they had trouble accessing the tools they normally use, including their health monitoring system and server access. With the number of vendors and customers involved, the initial calls were particularly chaotic.

In the end, it would take several days to figure out what had actually happened by reproducing the outage in their lab. The issue started with routine server maintenance on an OS that was different than most of the servers they ran. When that server rebooted, it put an LLDP packet out on the network. Due to a bug, CSG's network software picked it up and interpreted it as a spanning tree. It broadcast it out to the network and it was then picked up by their load balancer. Due to a misconfiguration in the load balancer, this got rebroadcast to the network, creating a network loop, taking the network down.

The aftermath was severe. The extent of angry customers required leadership to pivot their focus from their planned work (strategic initiatives, etc.) to just this outage. Throughout the company, there was also a large sense of loss and heartbreak over having failed their customers so severely. Morale was extremely low. Hurtful things were said, like “DevOps doesn’t work.”

CSG knew they wanted to respond to this failure differently. They needed to maximize learnings while also reducing the likelihood of an incident like this happening again. Their first step was incident analysis.

Their standard incident analysis was a structured process to help them understand what happened and identify opportunities for improvement. They did this by understanding the timeline of the incident; asking, “What happened? How can we detect it sooner? How can we recover sooner? What went well?” Understanding system behavior. And maintaining a blameless culture and avoiding finger pointing.

Because of this incident they also knew they needed to up their game. They reached out to Dr. Richard Cook and John Allspaw of Adaptive Capacity Labs to analyze the incident. Through two weeks of intense interviews and research, they gained a more thorough understanding of the events, and, in particular, learned the different perspectives of the people who were working on the outage.

From this intensive retrospective, they created an operation improvement program based on the Incident Command System. They broke the program into four categories: incident response, tool reliability, data center/platform resiliency, and application reliability.

Even before the whole organization had been through the training for their new incident management process, people started to see observable improvements in how outage calls were run: clutter on the calls had been removed; status reports had a known, steady cadence; and having an LNO (liaison officer) helped avoid interruptions on the incident calls.

The next biggest improvement was a sense of control over chaos. The simple act of having predictable cadences and patterns to follow helped everyone feel more confident and in control. It also allowed activities to run in parallel until the set time for a status update, allowing that activity to run without interruption.

Additionally, decision-making was updated from the old system by giving the incident commander clear command and authority so there's no question about who can make decisions.

Now, CSG has a stronger organizational ability to perform incident management. They've reinforced and broadened culture norms around safety and, most impactfully, implemented the incident management system that changed how they run outage calls.

In this case study, a blameless post-mortem (retrospective) led CSG to completely revamp the way they handle incidents. They directly applied their learnings regarding how they conduct their work, changing their culture and not blaming an individual or team.

Conclusion

To create a just culture that enables organizational learning, we have to re-contextualize so-called failures. When treated properly, errors that are inherent in complex systems can create a dynamic learning environment where all of the shareholders feel safe enough to come forward with ideas and observations, and where groups rebound more readily from projects that don't perform as expected.

Both retrospectives and injecting production failures reinforce a culture that everyone should feel both comfortable with and responsible for surfacing and learning from failures. In fact, when we sufficiently reduce the number of accidents, we decrease our tolerance so that we can keep learning. As Peter Senge is known to say, "The only sustainable competitive advantage is an organization's ability to learn faster than the competition."[46]

[*] In January 2013 at re:Invent, James Hamilton, VP and Distinguished Engineer for Amazon Web Services, said that the US-East region had more than ten data centers all by itself, and added that a typical data center has between fifty thousand and eighty thousand servers. By this math, the 2011 EC2 outage affected customers on more than half a million servers.[4]

† See Martin Fowler's article on circuit breakers for more on this: https://martinfowler.com/bliki/CircuitBreaker.html.

‡ This practice has also been called *blameless post-incident reviews* as well as *post-event retrospectives*. There is also a noteworthy similarity to the routine retrospectives that are a part of many iterative and Agile development practices.

§ We may also choose to extend the philosophies of *transparent uptime* to our post-mortem reports and, in addition to making a service dashboard available to the public, we may choose to publish (maybe sanitized) post-mortem meetings to the public. Some of the most widely admired public post-mortems include those posted by the Google App Engine team after a significant 2010 outage, as well as the post-mortem of the 2015 Amazon DynamoDB outage. Interestingly, Chef publishes their post-mortem meeting notes on their blog, as well as recorded videos of the actual post-mortem meetings.[15]

¶ Specific architectural patterns that they implemented included fail fasts (setting aggressive timeouts such that failing components don't make the entire system crawl to a halt), fallbacks (designing each feature to degrade or fall back to a lower quality representation), and feature removal (removing non-critical features when they run slowly from any given page to prevent them from impacting the member experience). Another astonishing example of the resilience that the Netflix team created beyond preserving business continuity during the AWS outage was that Netflix went over six hours into the AWS outage before declaring a Sev 1 incident, assuming that AWS service would eventually be restored (i.e., "AWS will come back . . . it usually does, right?"). Only after six hours into the outage did they activate any business continuity procedures.[35]