acmqueue

Fault Injection in Production

Making the case for resilience testing

John Allspaw, Etsy

When we build Web infrastructures at Etsy, we aim to make them resilient. This means designing them carefully so that they can sustain their (increasingly critical) operations in the face of failure. Thankfully, there have been a couple of decades and reams of paper spent on researching how fault tolerance and graceful degradation can be brought to computer systems. That helps the cause.

To make sure that the resilience built into Etsy systems is sound and that the systems behave as expected, we have to see the failures being tolerated *in production*.

Why production? Why not simulate this in a QA or staging environment? First, the existence of any differences in those environments brings uncertainty to the exercise, and second, the risk of not recovering has no consequences during testing, which can bring hidden assumptions into the fault-tolerance design and into recovery. The goal is to reduce uncertainty, not increase it.

Forcing failures to happen, or even designing systems to fail on their own, generally isn't easily sold to management. Engineers are not conditioned to embrace their ability to respond to emergencies; they aim to avoid them altogether. Taking a detailed look at how to respond better to failure is essentially accepting that failure will happen, which you might think is counter to what you want in engineering, or in business.

Take, for example, what you would normally think of as a simple case: the provisioning of a server or cloud instance from zero to production:

1. Bare metal (or cloud-compute instance) is made available.
2. Base operating system is installed via PXE (preboot execution environment) or machine image.
3. Operating-system-level configurations are put into place (via configuration management or machine image).
4. Application-level configurations are put into place (via configuration management, app deployment, or machine image).
5. Application code is put into place and underlying services are started correctly (via configuration management, app deployment, or machine image).
6. Systems integration takes place in the network (load balancers, VLANs, routing, switching, DNS, etc.).

This is probably an oversimplification, and each step or layer is likely to represent a multitude of CPU cycles; disk, network and/or memory operations; and various numbers of software mechanisms. All of these come together to bring a node into production.

Operability means that you can have confidence in this node coming into production, possibly joining a cluster, and serving live traffic seamlessly every time it happens. Furthermore, you want and expect to have confidence that if the underlying power, configuration, application, or compute resources (CPU, disk, memory, network, etc.) experience a fault, then you can survive such a fault by some means: allowing the application to degrade gracefully, rebuild itself, take itself out of production, and alert on the specifics of the fault, etc.

This confidence is typically built in a number of ways:

- **Hardware burn-in testing.** You can run extreme tests on the various hardware components in a node in order to confirm that none of them would experience faults at the onset of load. This may not be necessary or feasible in a cloud-compute instance.
- **Unit testing of components.** Each service can be easily tested in isolation, and the configuration can be checksummed to ensure it meets expectations.
- **Functional testing of integrations.** Each execution path (usually based on an application feature) can be explored with some form of automated procedure to assure expected results.

Traditionally, these sensible measures to gain confidence are taken before systems or applications reach production. Once in production, the traditional approach is to rely on monitoring and logging to confirm that everything is working correctly. If it is behaving as expected, then you don't have a problem. If it is not, and it requires human intervention (troubleshooting, triage, resolution, etc.), then you need to react to the incident and get things working again as fast as possible.

This implies that once a system is in production, "Don't touch it!"—except, of course, when it's broken, in which case touch it all you want, under the time pressure inherent in an outage response.

This approach isn't as fruitful as it could be, on a number of levels.

In the field, you need to prepare for undesirable circumstances. Power can get cut abruptly. Changes to the application or configuration can produce unforeseen behaviors, no matter how full the coverage of testing. Application behavior under various resource-contention conditions (think traffic spikes from news events or firehose-like distributed denial-of-service attacks) can have surprising results. This isn't a purely academic curiosity; these types of faults can (and will) affect production and, therefore, in Etsy's case, our sellers and our business. These types of events, however, are difficult to model and simulate with an accuracy that would inspire confidence in the system's behavior in the face of these problems.

The challenge is that Web systems (like many "complex" systems) are largely intractable, meaning that:

- To be fully described, there are many details, not few.
- The rate of change is high; the systems change before a full description (and therefore understanding) can be completed.
- How components function is partly unknown, as they resonate with each other across varying conditions.
- Processes are heterogeneous and possibly irregular.

In other words, while testing outside of production is a very proper approach, it's incomplete because some behaviors can be seen only in production, no matter how identical a staging environment can be made.

Therefore, another option must be added to the confidence-gaining arsenal: fault injection exercises sometimes referred to as GameDay. The goal is to make these faults happen in *production* in order to anticipate similar behaviors in the future, understand the effects of failures on the underlying systems, and ultimately gain insight into the risks they pose to the business.

Causing failures in complex systems isn't a new concept. Organizations such as fire departments have been running full-scale disaster drills for decades. Web engineering has an advantage over these types of drills in that the systems engineers can gather a massive amount of detail on any fault at an extremely high resolution, wield a very large amount of control over the intricate mechanisms of failures, and learn how to recover very quickly from them.

FAULT INJECTION

Constructing a GameDay exercise at Etsy follows this pattern:

1. Imagine a possible untoward event in your infrastructure.
2. Figure out what is needed to prevent that event from affecting your business, and implement that.
3. Cause the event to happen in production, ultimately to prove the noneffect of the event and gain confidence surrounding it.

The greatest advantage of a GameDay exercise is figuring out how to prevent a failure from affecting the business. It's hard to overstate the importance of steps 1 and 2. The idea is to get a group of engineers together to brainstorm the various failure scenarios that a particular application, service, or infrastructure could experience. This will help remove complacency about the safety of the overall system. Complacency is an enemy of resilience. If a system has a period of little or no degradation, there is a real risk of it drifting toward failure on multiple levels, because engineers can become convinced—falsely—that the system is experiencing no problems because it is inherently safe.

Imagining failure scenarios and asking, "What if…?" can help combat this thinking and bring a constant sense of unease to the organization. This sense of unease is a hallmark of high-reliability organizations. Think of it as continuously deploying a BCP (business continuity plan).

BUSINESS JUSTIFICATION

In theory, the idea of GameDay exercises may seem sound: you make an explicit effort to anticipate failure scenarios, prepare to handle them gracefully, and then confirm this behavior by purposely injecting those failures into production. In practice, this idea may not be appealing: it brings risk to the forefront, and without context, causing failures on purpose may seem crazy. What if something goes wrong?

The traditional view of failure in production is avoidance at all costs. The assumption is that failure is entirely preventable, and if it does happen, then find the persons responsible (usually those closest to the code or systems) and fire them, in the belief that getting rid of "bad apples" is how you bring safety to an organization.

This perspective is, of course, ludicrous. Fault injection and GameDay scenarios can produce a more pragmatic and realistic view.

When I approached Etsy's executive team with the idea of GameDay exercises, I explained that it's not that we want to cause failures out of some perverse need to watch infrastructure crumble; it's that we know that parts of the system *will* inevitably fail, and we need to gain confidence that the system is resilient enough to handle failure gracefully.

The concept, I explained to the executives, is that building resilient systems requires experience with failure, and that we want to anticipate and confirm our expectations surrounding failure *more* often, not *less* often. Shying away from the effects of failure in a misguided attempt to reduce risk will result in poor designs, stale recovery skills, and a false sense of safety.

In other words, it's better to prepare for failures in production and cause them to happen *while we are watching,* instead of relying on a strategy of hoping the system will behave correctly *when we aren't watching*. The worst-case scenario with a GameDay exercise is that something will go wrong during the exercise. In that case, an entire team of engineers is ready to respond to the surprises, and the system will become stronger as a result.

The worst-case scenario in the absence of a GameDay exercise is that something in production will fail that wasn't anticipated or prepared for, and it will happen when the team isn't expecting or watching closely for it.

How can you assure that injecting faults into a live production system doesn't affect actual traffic, revenue, and the end-user experience? This can be done by treating the fault-toleration and graceful degradation mechanisms as *features*. This means bringing all the other confidence-building techniques (unit and functional testing, staging hardware environments, etc.) to these resilience measures until you're satisfied. Just like every other feature of the application, it's not finished until you've deployed it to production and have verified that it's working correctly.

CASE: PAYMENTS SYSTEM

Earlier this year Etsy rolled out a new payment system (http://www.etsy.com/blog/news/2012/announcing-direct-checkout/) to provide more flexibility and reliability for buyers and sellers on the site. Obviously, resilience was of paramount importance to the success of the project. As with many Etsy features, the new system was rolled out to production in a gradual ramp-up. Sellers who wanted to use this new payment method could opt in, and Etsy turned the functionality on for groups of sellers at a time.

As you might imagine, the payment system is not particularly simple. It has fraud-detection components, audit trails, security mechanisms, processing-state machines, and other components that need to interact with each other. Thus, Etsy has a complex mission-critical system with very high expectations for resiliency.

To confirm its ability to withstand failures gracefully, Etsy put together a list of reasonable scenarios to prepare for, develop against, and test in production, including the following:

- One of the app servers dies (power cable yanked out).
- All of the app servers leave the load-balancing pool.
- One of the app servers gets wiped clean and needs to be fully rebuilt from scratch.
- Database dies (power cable yanked out and/or process is killed ungracefully).
- Database is fully corrupt and needs full restore from backup.
- Offsite database replica is needed to investigate/restore/replay single transactions.
- Connectivity to third-party sites is cut off entirely.

The engineers then put together all of the expectations for how the system would behave if these scenarios occurred in production, and how they could confirm these expectations with logs, graphs, and alerts. Once armed with these scenarios, they worked on how to make these failures either:

> —not matter at all (transparently recover and continue on with processing),
>
> —matter only temporarily (gracefully degrade with no data loss and provide constructive feedback to the user),
>
> —or matter only to a minimal subset of users (including an audit log for reconstructing and recovering quickly and possibly automatically).

After these mechanisms were written and tested in development, the time came to test them in production. The Etsy team was cognizant of how much activity the system was seeing; the support and product groups were on hand to help with any necessary communication; and team members went through each of the scenarios, gathering answers to questions such as:

- Were they successful in transparently recovering, through redundancy, replication, queuing, etc.?

- How long did each process take—in the case of rebuilding a node automatically from scratch, recovering a database, etc.?
- Could they confirm that no data was lost during the entire exercise?
- Were there any surprises?

The team was able to confirm most of the expected behaviors, and the Etsy community (sellers and buyers) was able to continue with its experience on the site, unimpeded by failure.

There were some surprises along the way, however, which the Etsy team took as remediation items after the exercise. First, during the payments process, a third-party fraud-detection service was contacted with information about the transaction. While Etsy uses a number of external APIs (fraud, device reputation, etc.), this particular service had no specified timeout on the external call. When testing the inability to contact the service, the Etsy team used firewall rules both to hard close the connection and to attempt to hang it open. Having no specified timeout meant that they were relying on the default, which was much too long at 60 seconds. The intended behavior was to fail open, which meant that the transaction could continue if the external service was down. This worked, but only after the 60-second timeout, which caused live payments to take longer than necessary during the exercise.

This was both a surprise and relatively easy to fix, but it was nonetheless an oversight that affected production during the test.

Recovering from database corruption also took longer than expected. The GameDay exercise was performed on one side of a master-master pair of databases, and while the recovery happened on the corrupted server, the remaining server in the pair took all reads and writes for production. While no production data was lost, exposure with reduced capacity occurred for longer than expected, so the Etsy team began to profile and then try to reduce this recovery time.

The cultural effect of the exercise was palpable. It greatly decreased anxiety about the ramp-up of the payments system; it exposed a few darker-than-desired corners of the code and infrastructure to improve; and it increased overall confidence in the system. Complacency is not an immediate threat to the system as a result.

LIMITATIONS

The goal of fault injection and GameDay exercises is to increase confidence in an otherwise complicated or complex system's ability to stay resilient, but they have limitations.

First, the exercises aren't meant to discover how engineering teams handle working under time pressure with escalating and sometimes disorienting scenarios. That needs to come from the postmortems of actual incidents, not from handling faults that have been planned and designed for.

The faults and failure modes are *contrived*. They reflect the fault designer's imagination and therefore can't be comprehensive enough to guarantee the system's safety. While any increase in confidence in the system's resiliency is positive, it's still just that: an increase, not a completion of perfect confidence. Any complex system can (and will) fail in surprising ways, no matter how many different types of faults you inject and recover from.

Some have suggested that continually introducing failures automatically is a more efficient way to gain confidence in the adaptability of the system than manually running GameDay exercises as an engineering-team event. Both approaches have the same limitation mentioned here, that they increase confidence but can't be used to achieve sufficient safety coverage.

Automated fault injection can carry with it a paradox. If the faults that are injected (even at random) are handled in a transparent and graceful way, then they can go unnoticed. You would think this was the goal: for failures not to matter whatsoever when they occur. This masking of failures, however, can result in the very complacency that they are intended (at least should be intended) to decrease. In other words, when you've got randomly generated and/or continual fault injection and recovery happening successfully, care must be taken to *raise* the detailed awareness that this is happening—when, how, where, etc. Otherwise, the failures themselves become another component that increases complexity in the system while still having limitations to their functionality (because they are still contrived and therefore insufficient).

FEAR

A lot of what I'm proposing should simply be an extension of the confidence-building tools that organizations already have. Automated quality assurance, fault tolerance, redundancy, and A/B testing are all in the same category of GameDay scenarios, although likely with less drama.

Should everything have an associated GameDay exercise? Maybe, or maybe not, depending on your level of confidence in the components, interactions, and levels of complexity found in your application and infrastructure. Even if your business doesn't think that GameDay exercises are warranted, however, they ought to have a place in your engineering toolkit.

SAFETY VACCINES

Why would you introduce faults into an otherwise well-behaved production system? Why would that be useful?

First, these failure-inducing exercises can serve as "vaccines" to improve the safety of a system—a small amount of failure injected to help the system *learn* to recover. It also keeps a concern about failure alive in the culture of engineering teams, and it keeps complacency at bay.

It gathers groups of people who might not normally get together to share in experiencing failures and to build fault tolerance. It can also help bring the concept of operability in production closer to developers who might not be used to it.

At a high level, production fault injection should be considered one of many approaches used to gain confidence in the safety and resiliency of a system. Like unit testing, functional testing, and code review, this approach is limited as to which surprising events it can prevent, but it also has benefits, many of which are cultural. We certainly can't imagine working without it.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

JOHN ALLSPAW is senior vice president of tech operations at Etsy. He has worked in systems operations for more than 14 years in biotech, government, and online media. He started out tuning parallel clusters running vehicle crash simulations for the U.S. government and then moved on to the Internet in 1997. He built the backing infrastructures at Salon, InfoWorld, Friendster, and Flickr. He is the author of *The Art of Capacity Planning* (O'Reilly Media, 2008) and *Web Operations* (O'Reilly, 2010).