The InfoQ eMag / Issue #77 / October 2019

Taming Complex Systems in Production

@emilywithcurls

InfoQ

FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

InfoQ @ InfoQ InfoQ InfoQ

Taming Complex Systems in Production

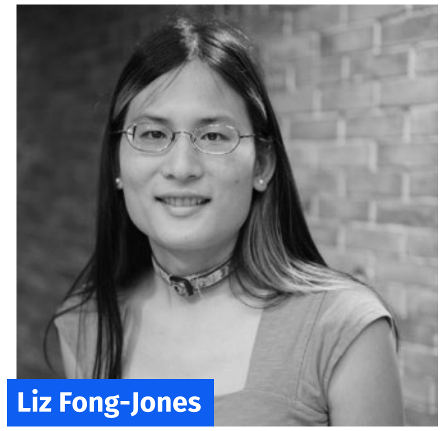
IN THIS ISSUE

PRODUCTION EDITOR Ana Ciobotaru / COPY EDITORS Lawrence Nyveen & Susan Conant / DESIGN Dragos Balasoiu / COVER ILLUSTRATION by Emily Griffin (@emilywithcurls)

GENERAL FEEDBACK feedback@infoq.com / ADVERTISING sales@infoq.com / EDITORIAL editors@infoq.com

CONTRIBUTORS

Nicky Wrightson is currently a principal engineer working on the data platform at Skyscanner, the world's travel search engine. Prior to this Nicky worked as a principal engineer at the Financial Times, a media organisation - here she led the roll out of a new content and metadata platform with a revamped operational model alongside it. She has a passion for driving change in more than just the creation of new cloud native architectures but also the necessary cultural evolution that is necessary.

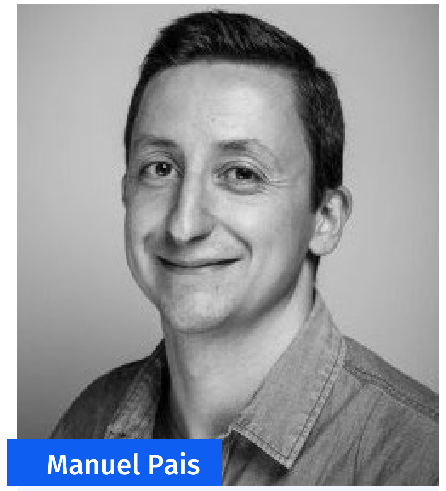

Liz Fong-Jones is a developer advocate, labor and ethics organizer, and site-reliability engineer with more than 15 years of experience. She is an advocate at Honeycomb.io for the SRE and Observability communities, and previously was a SRE working on products ranging from the Google Cloud Load Balancer to Google Flights.

Lubos Parobek leads product management and user interaction at Sauce Labs, provider of the world's most comprehensive and trusted continuous testing cloud for web and mobile applications. His previous experience includes product leadership positions at organizations including KACE, Sybase iAnywhere, AvantGo, and 3Com. Parobek holds a Master of Business Administration from the University of California, Berkeley

Daniel Bryant works as an Independent Technical Consultant and Product Architect at Datawire. His technical expertise focuses on 'DevOps' tooling, cloud/container platforms, and microservice implementations. Daniel is a Java Champion, and contributes to several open-source projects. He also writes for InfoQ, O'Reilly, and TheNewStack, and regularly presents at international conferences such as OSCON, QCon, and JavaOne. In his copious amounts of free time, he enjoys running, reading and traveling.

Michael Bryzek is the CTO and co-founder of Gilt Groupe, an innovative online shopping, destination offering its members special access to the most inspiring merchandise, culinary offerings, and experiences every day, many at insider prices. He has built Gilt's technology platform to support massive bursts of traffic, as sales start every day at noon.

A LETTER FROM THE EDITOR



Manuel Pais

is a DevOps and Delivery Consultant, focused on teams and flow. Manuel helps organizations adopt test automation and continuous delivery, as well as understand DevOps from both technical and human perspectives. Co-curator of DevOpsTopologies.com. DevOps lead editor for InfoQ. Co-founder of DevOps Lisbon meetup. Coauthor of the book "Team Topologies: Organizing Business and Technology Teams for Fast Flow." Tweets @manupaisable

Software systems are becoming more complex, outages are becoming more expensive, and consumers are becoming less tolerant of downtime. All of this typically exerts a high mental (and physical) strain on operations engineers. Acknowledging the fragility of complex systems is the first step in building resilience into systems and people. Further, to tame complexity and its effects, organizations need a structured, multi-pronged, human-focused approach, that: makes operations work sustainable, centers decisions around customer experience (tip: that's what SLOs are for), uses continuous testing (yes, including in production), and includes chaos engineering and system observability. In this eMag, we cover all of these topics to help you tame the complexity in your system and create a healthy working environment for people handling production.

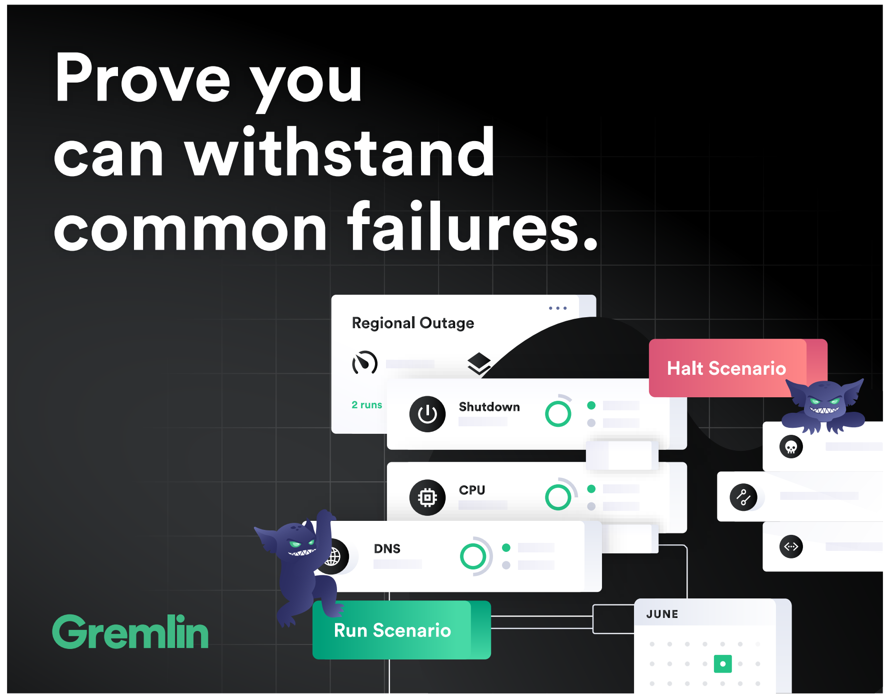
Nicky Wrightson brings five tried and true techniques that helped her team at *The Financial Times* nearly eliminate out-of-hours support calls. This is not to say their system was fault-free, but rather that they were able to build a resilient technical and operational system. Wrightson highlights the importance of having teams that own their system, not just from a delivery point of view, but also the operational model and all the aspects supporting it. For instance, regularly practicing (injecting) failure scenarios recovery is fundamental to increase confidence of those supporting the system.

Speaking of failure scenarios and recovery, Daniel Bryant's Q&A with prominent early adopters of chaos engineering from multiple organizations gives an overview of the benefits, challenges, and practices in this critical area for companies investing in seriously improving and learning from incident response. Reading this piece will help you separate the myth (chaos engineering is running random attacks in production) from the truth (chaos engineering is a principled practice of experimentation and information sharing that includes testing in production).

Liz Fong-Jones stresses that production ownership (putting the team who develops a service on call) is not equivalent to production excellence. Without proper training and safeguards to ensure people's well-being, production ownership can actually have a negative effect on service reliability and, even worse, demoralize the team.Tooling helps automate processes, but teams also need a roadmap for developing the necessary skills for production excellence. These include measuring what matters (SLOs), observability-enabled quick diagnosis of unknown issues, inter-team collaboration, blameless retrospectives, and risk analysis.

Continuous testing is critical to assess the quality and reliability of systems, among other ilities. But without a strong focus on the quality of the tests, testing can become costly and ineffective, slowing down delivery. Lubos Parobek shares four key practices to ensure high test quality: investing in a high pass-rate, keeping tests short and atomic (which leads to shorter test execution and more reliable tests), testing across multiple platforms, and leveraging parallelization (to ensure test suites can grow without compromising speed of execution).

What if we could do continuous testing reliably and safely, not only during delivery but also in production? While chaos engineering focuses on failure injection, Michael Bryzek's talk (of which we include a summary) focuses on testing happy day scenarios in production, dramatically increasing the confidence that business critical services are working as expected every day, every hour, and every minute. Bryzek illustrates with practical examples how implementing safeguards to testing in production can be much simpler than most people expect.What tends to be more challenging is architecting and delivering distributed services with minimal dependencies and learning to develop and trust high-quality automated tests.



An Engineer's Guide to a Good Night's Sleep

by **Nicky Wrightson**, Principal Engineer at Skyscanner

We are building really complicated distributed systems. Increased microservices adoption, fueled by the move into the cloud where architectures and infrastructure can flex and be ephemeral, adds complexity every day to the systems we create and maintain. All of this complexity takes place alongside operating models with autonomous and totally empowered teams, so each distributed system has its own tapestry of technical approaches, languages, and services.

Thankfully, despite such complications, you can build a technically complex, yet supportable system, using a set of best practices gathered from my experience building a new generation of APIs in a completely greenfield manner.

Complexity is the Cost of Flexibility

Microservices are autonomous and self-contained, and allow us to deploy independently without being shackled to lengthy release coordination. Being cloud native allows us a lot more flexibility around scaling. But, these new approaches and technical capabilities are used for organisational reasons, such as scaling teams while still maintaining momentum, not to reduce technical complexity. So, all of this comes at an operational cost and a rise in complexity.

Martin Fowler talks of the trade-offs of microservices, with one being:

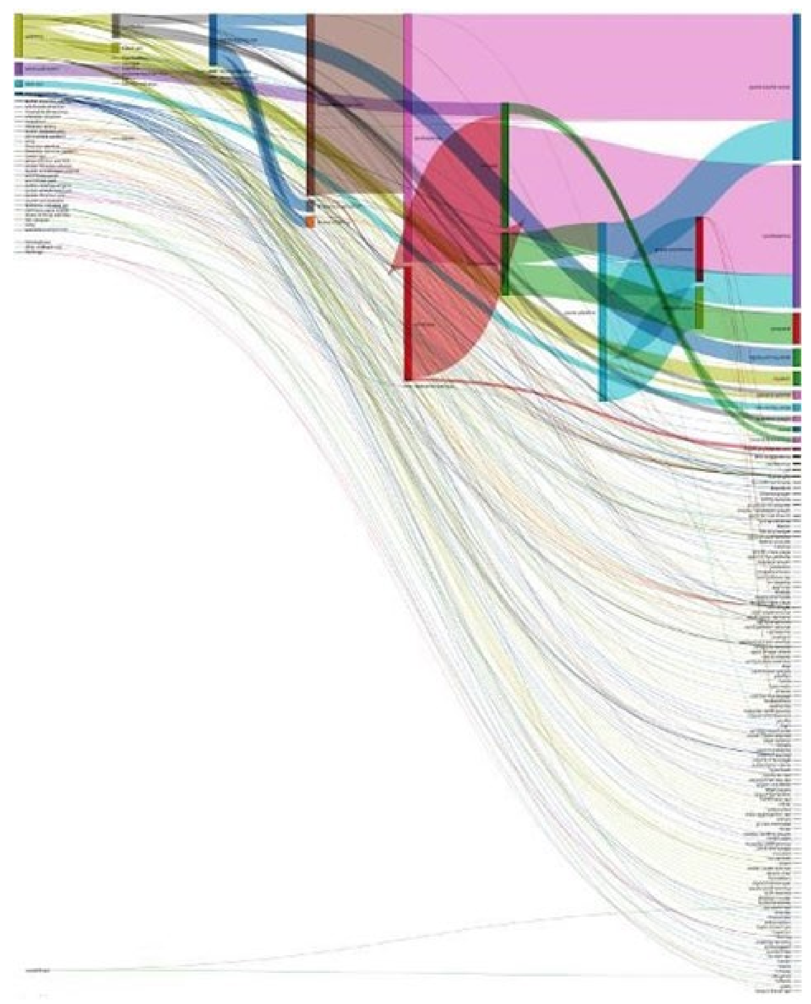
"Operational Complexity: You need a mature operations team to manage lots of services, which are being redeployed regularly."

While working for the Financial Times, I had the opportunity to help a team build a new generation of content APIs in a complete greenfield manner. We were empowered - a self-organised team that could choose how to re-architect, deploy, and maintain all this new functionality.

We could choose all aspects of the tech stack, but we also had to define our support model—we were fully accountable and we knew it, so we built with that in mind. This accountability made us view the operational-support model differently than on previous projects.

Initially, our implementation was very flaky with APIs regularly unavailable—or worse—still serving unreliable data. This drove our initial major consumer, a new version of the ft.com site, to guard against us by using caches to store our data in case the API was down. When it came to the go-live of this new site, they wanted assurances—service-level agreements (SLAs), from us.

The most important thing to the Financial Times is breaking news. We were therefore asked for a very tight SLA of 15 minutes recovery for the APIs involved in breaking news. Now, this is pretty tough, especially as we had some pretty big datastores behind the scenes. This SLA not only brought with it a set of technical challenges, but also several organisational challenges, such as out-of-hours support. Luckily for us, we were empowered to define how to do that in the way that best suited us. We did do it and managed to virtually eliminate out-of-hours calls.



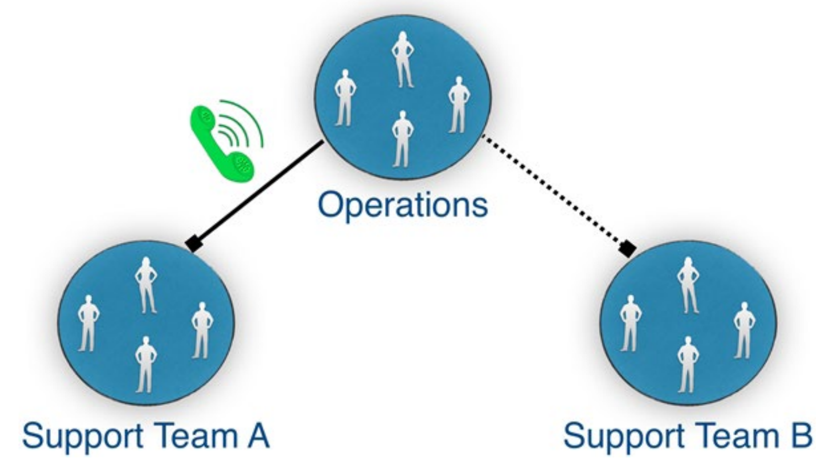
Skyscanner service tracing

We didn't build a mythical, magical system that was fault-free, but we did build a resilient technical and operational system. Five best practices brought us to that point of resilience:

1. Enable the Engineer

This is ultimately the most important part of building a resilient system. Every engineer must start thinking of operability as a primary concern, not as an afterthought. They need to start questioning their implementations continuously—"What do I want to happen if there is an error: Get called regardless of the time? Retry? Log something to look at later (perhaps)?"

We are now working in or at least moving toward, autonomous, empowered teams or squads. This approach gives us accountability, but also the influence over the support model we want to use. It doesn't matter if you



The support model defined by Content Programme at the Financial Times

operate in the traditional model of a dedicated company-wide, first-line support team, or teams that are called directly from tools like pager duty.

Of course, working with other teams complicates the process, but can also take the pressure off the engineers—it is a trade-off and often depends on the company-wide model. Teams need to define how they work together best and ensure they have adequate tooling. Above all, for the teams to work effectively, they need to foster and evolve the relationships within each team and among collaborating teams.

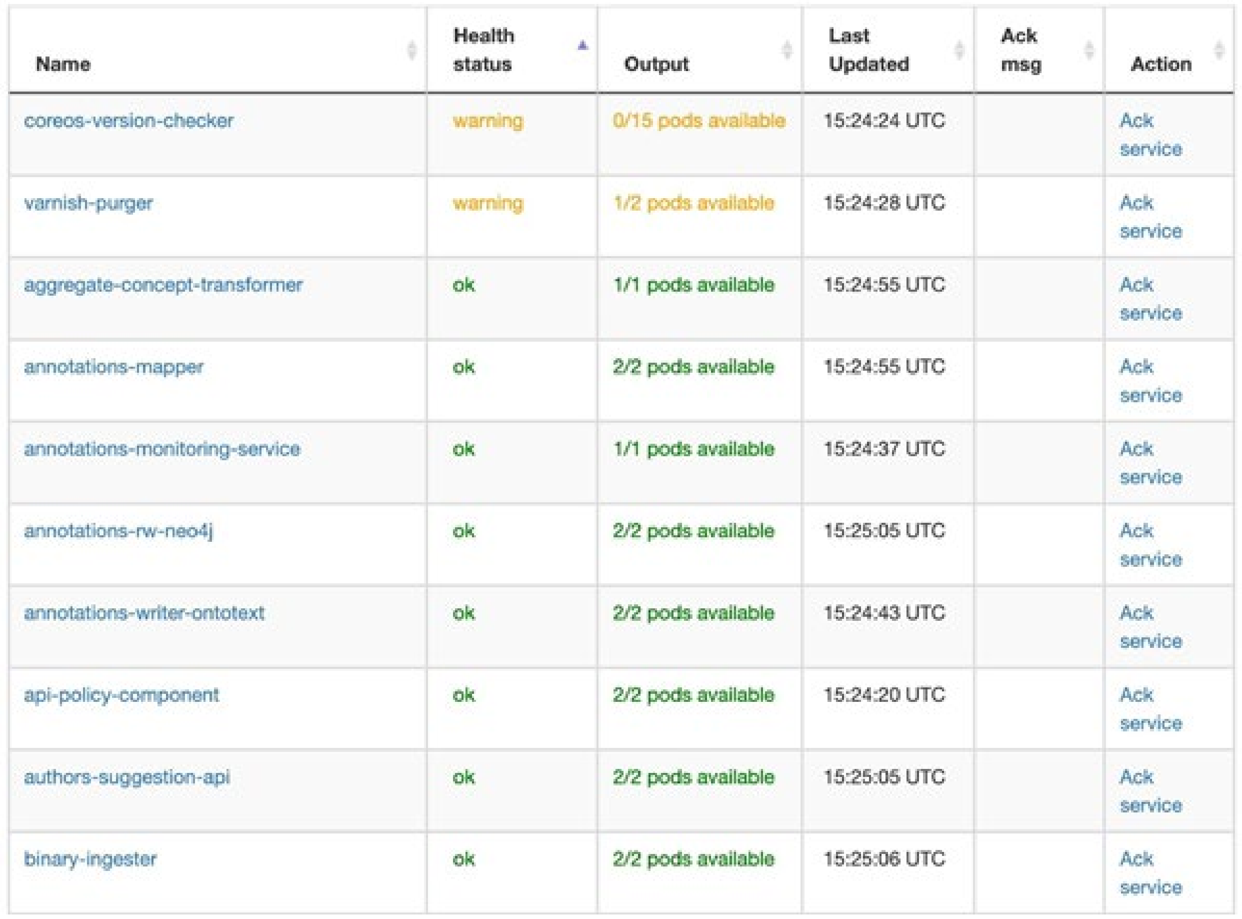
Rethinking Out-of-hours Support
When the SLA changed for the content APIs with the new site rollout, there had to be commitments about out-of-hours support. We took the opportunity to examine the way we did this. We were keen to reduce the effect of out-of-hours support on people's home life. We achieved this using two buckets of engineers, as shown in the following figure.

The operations team could round robin call the engineers in the bucket, with one bucket being "on call" from week to week. The engineers were not obligated to answer the call, and if they didn't, then the next person would get called. This allowed people to spontaneously head to the pub, out for a bike ride, or swim, without worrying they might get called. This system was completely voluntary and well compensated—people got overtime if they answered a call. It is important to recognise that some people are just unable to commit to this, and also important to recognise this is not part of the day-to-day job, so they should be paid.

This support model extended into the daytime also. There were two people who responded to incidents, queries, and also worked to improve the platform and operational processes. They rotated in from the engineering teams. This was a dedicated role because we recognised this as a critical function and no operational improvements would happen if they were working on scheduled work. Additionally, this meant engineers were raising pull requests across the entire estate, not only the areas with which they were familiar. This platform-wide issue handling was an important aspect. It helped bolster the confidence of the engineers to cope with calls out of hours and encouraged them to think of operational concerns when implementing business functionality. The pain of bad practices would bite them while on rotation, or worse still, at 3 am.

People design systems differently if they understand their code could result in an out-of-hours call. "Will this error affect the business?" For the Financial Times, the question was always, "Will this error affect the brand?" But, that business could be a whole lot worse if you worked for a hospital or power station!

```
if err != nil {
    // Does this error mean I get called at 3am?
}
```



Name	Health status	Output	Last Updated	Ack msg	Action
coreos-version-checker	warning	0/15 pods available	15:24:24 UTC		Ack service
varnish-purger	warning	1/2 pods available	15:24:28 UTC		Ack service
aggregate-concept-transformer	ok	1/1 pods available	15:24:55 UTC		Ack service
annotations-mapper	ok	2/2 pods available	15:24:55 UTC		Ack service
annotations-monitoring-service	ok	1/1 pods available	15:24:37 UTC		Ack service
annotations-rw-neo4j	ok	2/2 pods available	15:25:05 UTC		Ack service
annotations-writer-ontotext	ok	2/2 pods available	15:24:43 UTC		Ack service
api-policy-component	ok	2/2 pods available	15:24:20 UTC		Ack service
authors-suggestion-api	ok	2/2 pods available	15:25:05 UTC		Ack service
binary-ingester	ok	2/2 pods available	15:25:06 UTC		Ack service

The severity level is clearly stated in the aggregated system-health check.

Handling Errors
Our next improvement was hooking up the errors with alerting in a sensible manner. Thinking about severity levels is a useful tool to differentiate between something that can be fixed during normal hours. Not all errors are going to cause issues to the primary functionality of the system. An example of what we adopted is show above.

We had a containerised platform, starting with one we hand-rolled ourselves before migrating to Kubernetes. We wanted to understand immediately when our platform was unable to perform critical business activities such as publishing content. However, there were some "not-working" services that could be recovered in-hours, at our leisure. The example above shows a health-check page that aggregated the health of the entire platform and if the aggregate went to critical, we got called. However, in the example shown, we are warned we have an out-of-date version of CoreOS, the container operating system we used. This warning was very important to address because it could have critical security patches; but, it does not need to be done at 3 am on a Sunday morning. It can be addressed post-coffee on a Monday instead.

Engineers can address operational issues going forward, as well as glance back and keep updating older applications as the operational manner is refined.

Lehman's laws of Software Evolution states:

> **"The quality of a system will appear to be declining unless it is rigorously maintained."**
>
> **"As a system evolves, its complexity increases unless work is done to maintain or reduce it."**

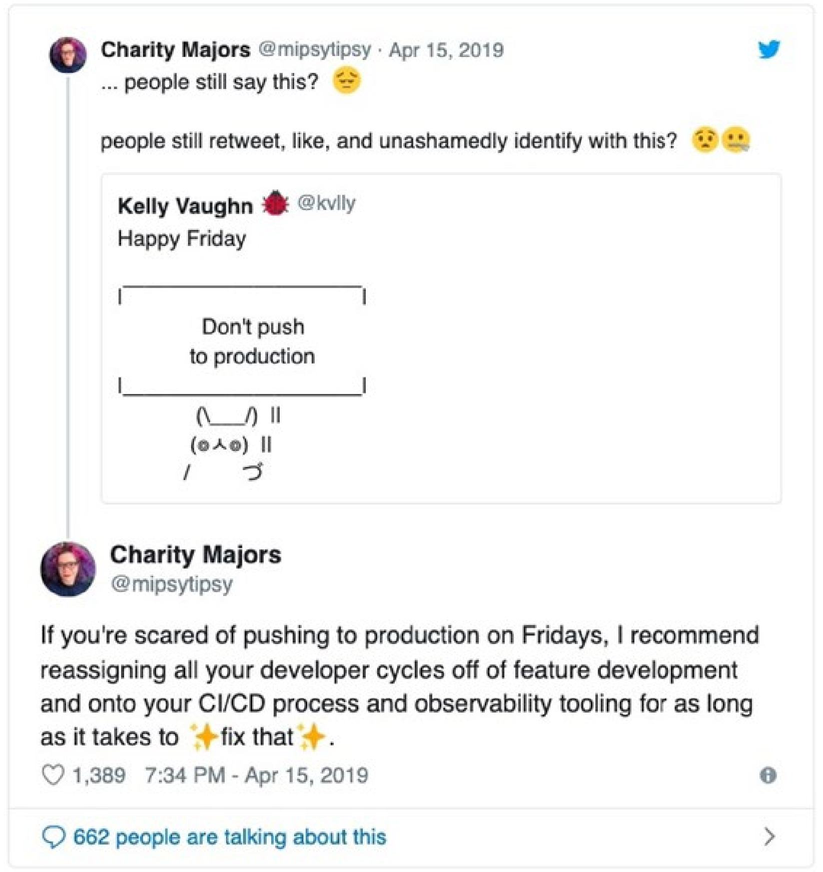
This can apply to technical implementations and the operation-support model. Engineers need to maintain and enhance the operational aspects of the system to keep it functioning as required.

2. Avoid out-of-hours calls, by catching more in-hours calls

This is probably the largest category, responsible for getting one of those out-of-hours calls. However, there are a few things you can do to reduce this likelihood.

Releases

Releases are the biggest culprit. Even with the best will in the world, we can't completely guarantee a problem-free release. Scenarios could lay undiscovered—a slow "leak" of an issue that becomes more noticeable over hours, days, or weeks. Releases are always going to increase the chance of getting called, but we can do some things to reduce that risk.



Charity Majors @mipsytipsy · Apr 15, 2019
... people still say this? 😒

people still retweet, like, and unashamedly identify with this? 😟😩

Kelly Vaughn 🐞 @kvlly
Happy Friday

```
|‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾|
   Don't push
  to production
|________________|
  (\__/) ||
  (•ㅅ•) ||
  /    づ
```

Charity Majors
@mipsytipsy

If you're scared of pushing to production on Fridays, I recommend reassigning all your developer cycles off of feature development and onto your CI/CD process and observability tooling for as long as it takes to ✨fix that✨.

♡ 1,389 7:34 PM - Apr 15, 2019

662 people are talking about this

From "Friday Deploy Freezes Are Exactly Like Murdering Puppies" blog post by @mipsytipsy

It's not a solution to not release at 5 pm. Fear of late-day releases is probably a good indicator that there is low confidence in the release or with the team supporting the release. Of course, there is always a balance of risk:

- What is the impact of release?
- How quickly could this be rolled back?
- How confident am I that this release isn't going to break things?

Charity Majors blogs about this, likening Friday freezes to "murdering puppies:"

Deployments should be as quick as possible; don't underestimate peoples' attention span. If it takes a long time, people will wander off and make a cup of tea or worse still, head home, before verifying the release. If you are unable to make your deployments super quick, then make sure you get poked to verify the release has gone out successfully. A simple Slack alert hooked into your CI would suffice. Alternatively, you can automate the verification.

Once in production, verification in some manner is critical and does not necessarily need to be sophisticated. This means testing in production, which is not as scary as it may sound. I don't mean for anyone to throw their code into production and hope for the best. Some options:

- Have production details and be able to perform a manual test
- Use feature flags with a small number of users exposed to the change initially
- Turn off the new feature overnight, while you build confidence in the release
- Constantly replay requests to highlight any issues early on

James Governor called this "Progressive Delivery" and Cindy Sridharan also talks of testing in production.

Besides releases, the other thing that can massively increase your chances of a call in the middle of the night—batch jobs.

Batch Jobs

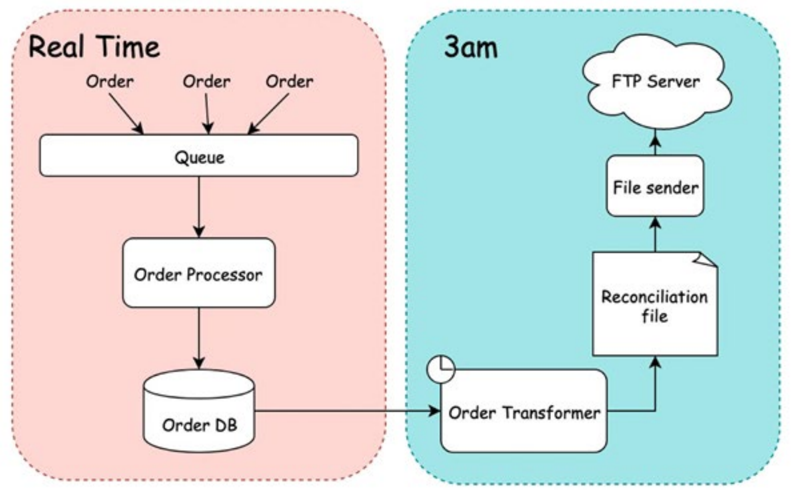
Batch jobs that do all the heavy lifting of a job in the middle of the night mean you get delayed feedback for an issue, at exactly the time you don't want it.

The adjacent figure illustrates this.

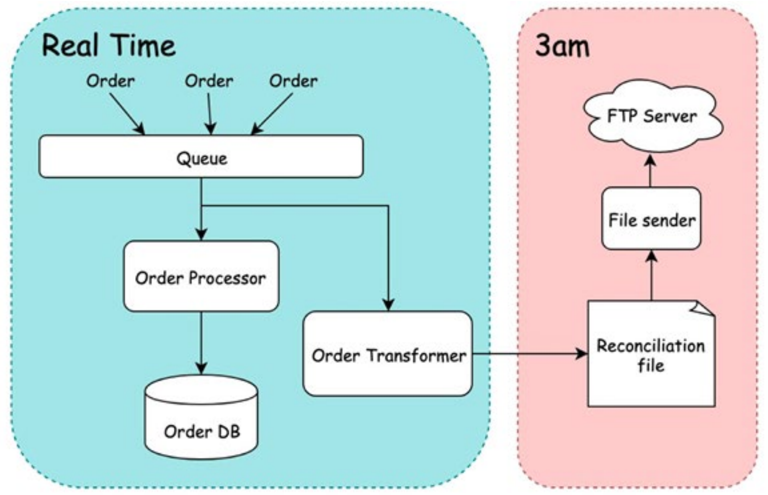
Here we have a simple order system that is getting orders in an event-driven manner. However, it aggregates and transforms all those orders at 3am, before sending a file off to third-party reconciliation.

If we could move much of the heavy computation back into the day, we shift the call back into business hours, shown below:

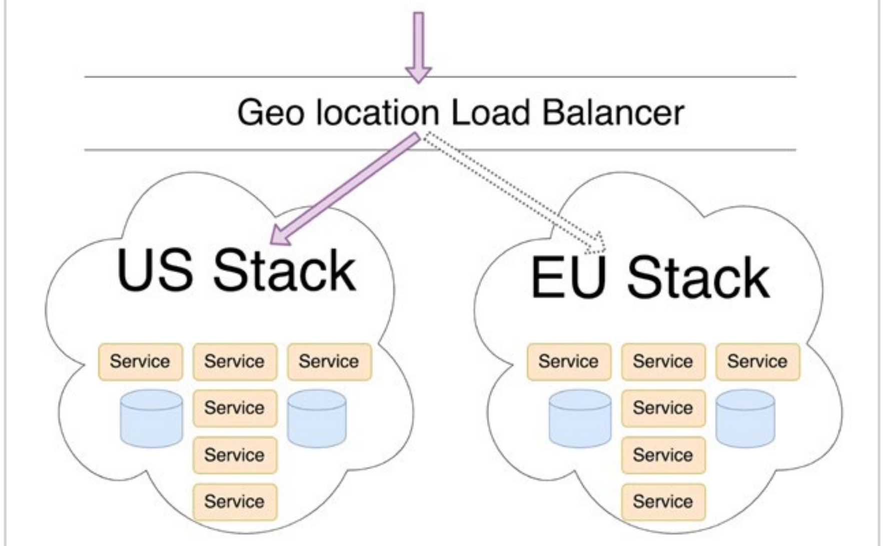
So, if we transformed or aggregated the orders in real-time, the job at 3 am merely has to FTP the file off to a third party.



An order system that takes real-time order, but transforms them at 3am



An improved order system that moves the transformation aspect to real-time



An ACTIVE/ACTIVE multi region full system deployment

3. Automate failure recovery wherever possible

Don't get woken up for something a computer can do for you; computers will do it better anyway.

The best thing to come our way in terms of automation is all the cloud tooling and approaches we now have. Whether you love serverless or containers, both give you a scale of automation that we previously would have to hand roll.

Kubernetes monitors the health checks of your services and restarts on demand; it will also move your services when "compute" becomes unavailable. Serverless will retry requests and hook in seamlessly to your cloud provider's alerting system.

These platforms have come a long way, but they are still only as good as the applications we write. We need to code with an understanding of how they will be run, and how they can be automatically recovered.

Develop applications that can recover without humans

Ensure that your applications have the following characteristics:

- Graceful termination: Terminating your applications should be considered a norm, not an exception. If you lose a node of your Kubernetes cluster, then Kubernetes will need to schedule your application on a different VM.
- Transactional: Make sure your application can fail at any point and not cause half-baked data.
- Clean restarts: We are guaranteed to lose VMs in the cloud, and we need to withstand this— whether it's a new function spinning up or a container being moved.
- Queue backed: One way you can get your applications to pick up where they left off if suddenly terminated—back them up via a queue
- Idempotency: This means that repeated requests will have the same outcome, and is really useful for being able to replay failed events.
- Stateless: Stateless microservices are much easier to move around; so, where possible, make them stateless.

Allow for the possibility of complete system failure

There are also techniques for dealing with situations when an outage is greater than one service, or if the scale of the outage is not yet known. One such technique is to have your platform running in more than one region, so if you see issues in one region, then you can failover to another region.

For example, at the Financial Times, we had an ACTIVE/ACTIVE setup, where we had our entire cluster running both in the EU and in the US, and requests were geo-location routed to either cluster, as shown in the diagram on the previous page.

We also had an aggregated health check for each cluster, and if any one of them went critical, we would direct all traffic to the other cluster. On recovery, it would automatically fail back to serving from both. If you have an ACTIVE/PASSIVE setup, you might need to script some state recovery/verification into your fail back options.

4. Understand what your customers care about

Understanding what your customers really care about is not as straightforward as you would first think, because if you ask them, they will say "everything!" When you understand your domain, however, you start to understand the critical nature of certain functionality. For example, for the Financial Times, their brand is utterly critical and linked to that is the necessity to be able to break the news, so while working on the content platform we viewed the publication and ability to read content as critical.

Once you understand the critical flows, you need to know when things are broken BEFORE your customer. Don't let your customers be the ones to inform you of outages or issues. You want to make sure you alert on critical flows without additional noise.

"Only have alerts that you need to action." - Sarah Wells, Director of Operations and Reliability @ the Financial Times

"Alert fatigue" means that if you alert on every error, you become almost blind to those alerts. Only have alerts for things that you will take action on right away, even if that alert is for an error you have no plans to do anything about because you can address those in the future if/when they become important to the system. You have logs you can query in the future for those unalerted issues if you want to fix them. After all, there is no such thing as a perfect system.

Not all applications are equal. Only alert on business-critical apps that affect those flows you've previously identified. Some breaks need a call, while others can wait until you have a coffee in the office the following day.

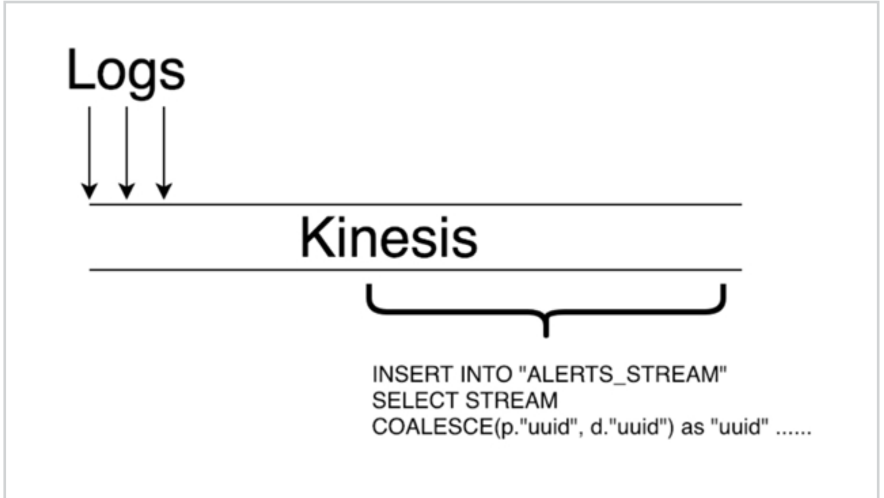
Having alerts in place is great, but you don't necessarily want to wait until there is a real failure to send an alert, and this is particularly true if you have spiky traffic. A content platform is a really great example of this. The Financial Times is global, but most content is written inside the UK, so in the early hours of Sunday there will be very few publishes, if any. So, this situation presents a "Schrodinger's platform"—is the platform dead or have there been no publishes?

To ensure we had a steady flow of feedback on our critical flows, we introduced something called synthetic requests, or simply, republishes of content we pumped through the system continuously. This enabled us to alert if something broke with an actual publish being involved.

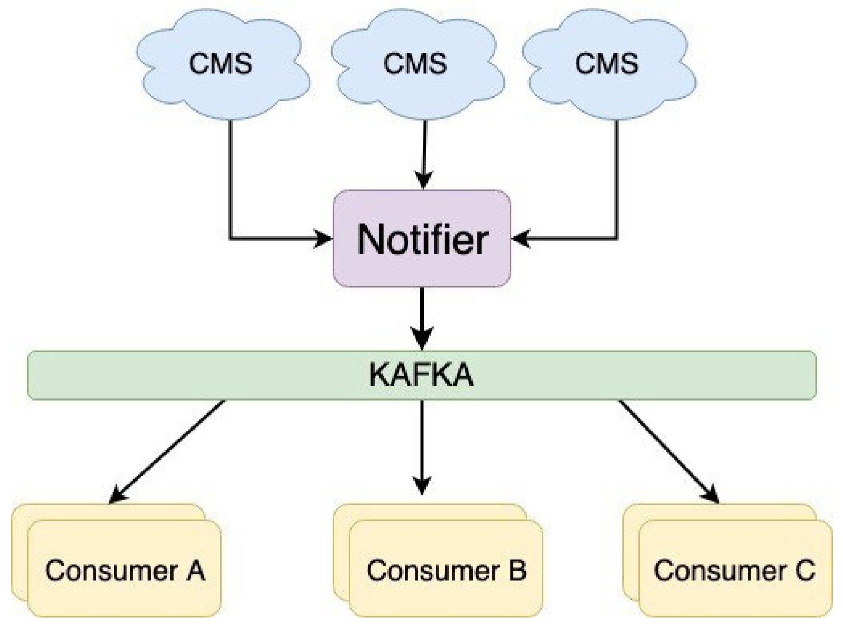
These synthetic requests gave us some amazing additional benefits—we deployed monitoring and these synthetic requests in lower environments, so we actually ended up using our monitoring as end-to-end tests also.

We instrumented our code with rudimentary tracing, by just embedding a transaction identifier. There is now a plethora of tooling such as Zipkin that can provide much richer tracing information. Ben Sigelman did a great keynote at QCon London 2019 on the importance of tracing: "Restoring Confidence in Microservices: Tracing That's More Than Traces."

This tracing allowed us to track our publishes through the system and react when they got stuck somewhere along the line. We initially wrote a service that did the monitoring, but it became quite brittle and one needed to understand a lot of business logic dispersed across the system, which seemed like an anti-pattern in the microservices world. So, we moved toward using structured logging to identify monitoring events. We could then


Logs
Kinesis
INSERT INTO "ALERTS_STREAM"
SELECT STREAM
COALESCE(p."uuid", d."uuid") as "uuid" ......

Analysis of logs using streaming SQL on a kinesis stream



A simplified view of the Content Platform at the Financial Times

feed those logs into Kinesis and run streaming SQL over the results, as shown above:

This approach meant we could ask, "Have we got all the expected results within a certain timeframe?"

5. Break things and practice

After working through the previous best practices, it may seem you've done everything possible to not get called, but what do you do if all else fails and you do get that 3 am call?

"Chaos engineering **is the discipline of experimenting on a software system in production in order to build confidence in the system's capability to withstand turbulent and unexpected conditions."**

We can test how resilient our system is to failure by pulling bits and pieces out or down. This can be done at the system level, but can also be very useful in testing the human element of operational support.

When should you release the chaos monkeys? Everyone is at a different point in the journey of their system's maturity, so the answer—it depends. However, if you are doing the fairly common journey of migration from a monolith to microservices, then when do you want to start experimenting? You obviously can't do this when you still have your monolith, because you would take down the whole system. However, I don't think you have to wait until you are at the very end of your journey to do this style of experimentation. Equally, you don't need to implement anything automated or elaborate to get value from such experimentation. You can manually take services or VMs down and see how the system reacts.

There are other ways to practice supporting the system, too. On the content platform at the Financial Times, there was a single point of failure at the entry for publishes, as shown below:

This single point of failure was definitely something we wanted to fix, but the impact was quite wide-reaching, so we hadn't managed to allocate some effort to it. It was also something that didn't cause as much pain as you would expect. In fact, it actually gave us some benefit—when we wanted to release an update to it, we would have to fail over to the other region while we released.

So, in effect, we were practicing failover quite regularly. If you rely on a mechanism for urgent support processes, make sure you practice it regularly to ensure it continues to work and also to reduce the fear and uncertainty around doing this. Following practices like these below will be your best bet for a good night's sleep:

- **Build teams that understand what being on call means, and as they own the system, they also own all the aspects of supporting it.**
- **Ensure that your development process adheres to best practices to reduce the risk of causing calls from standard daytime activities.**
- **Automation of failure recovery is becoming increasingly straightforward to achieve, so utilize it as much as possible.**
- **Deeply understanding your customers' needs means you can react to failure more quickly.**
- **Don't wait until a catastrophic failure to test your responses—put processes and approaches in place you can use to test every aspect of your support.**

Regular practice increases confidence in supporting the system, which in turn increases the confidence to know what to do if you're called at 3 am.

TL;DR

- Support for highly available systems must adapt as architectural models and team organisation evolve.
- System ownership means we need to examine more than business deliverables.
- Build a technically complex, yet supportable system, using five best practices.
- Rethinking support, automating, catching errors early, and understanding what your customers really care about, and a solid mix of breaking things and practicing is the best hope for a good night's sleep.

Chaos Conf Q&A

Designing Chaos Experiments, Running Game Days, and Building a Learning Organization

by **Daniel Bryant**, Independent Tech Consultant | Consulting CTO | InfoQ Editor

At the occasion of the 2019 Chaos Conf, InfoQ sat down with a number of the presenters to discuss topics such as the evolution and adoption of chaos engineering, important learning related to the people and process aspects of running chaos experiments, and the biggest obstacles to mainstream adoption.

Readers looking to continue their learning journey with chaos engineering can find a summary of the recent Chaos Community Day v4.0, a multi-article chaos engineering eMag, and a summary of various QCon talks on resilient systems.

InfoQ: Could you briefly introduce yourself please?

Jason Yee: Hi, I'm a senior technical evangelist at Datadog. Datadog is a SaaS-based observability platform that enables developers, operations, and business teams to get better insight into user experiences and application performance.

Caroline Dickey: Hi! I'm a site-reliability engineer at Mailchimp, an all-in-one marketing platform for small businesses. I build tooling and configuration to support engineer momentum, and develop monitoring and service-level objectives in order to promote the health of our application, and I lead the chaos-engineering initiative at Mailchimp.

Joyce Lin: Hey! I'm a developer advocate at Postman—an API development platform used broadly in the development community. I work with organizations that are pioneering better software development practices.

Robert (Bobby) Ross: My name is Robert Ross, but people like to call me Bobby Tables after the popular XKCD comic. I'm a bleeding-edge enthusiast for technology, so I like the pain of trying new things. I'm the CEO of FireHydrant, an incident response tool.

Kolton Andrus: I'm CEO and co-founder of Gremlin. I cut my teeth on building reliable systems at Amazon, where my team was in charge of the retail website's availability. I built their first "chaos engineering" platform (that name hadn't been coined yet) and helped other teams perform their first experiments. Then I joined Netflix, who had just started blogging on this topic. I had the opportunity to build their next generation of fault-injection testing (FIT), and worked with all of the critical streaming teams to improve our reliability from three nines to four nines of uptime (99.99%). After seeing the value to Amazon and Netflix, I felt strongly that everyone would need this approach and

The Interviewees

Jason Yee

is a technical evangelist at Datadog, where he works to inspire developers and ops engineers with the power of metrics and monitoring. Previously, he was the community manager for DevOps & Performance at O'Reilly Media and a software engineer at MongoDB.

Robert Ross

(Bobby Tables) is FireHydrant's Founder and CEO.

Subbu Allamaraju

is vice-president at the Expedia Group.

Caroline Dickey

is a site-reliability engineer at Mailchimp in Atlanta, where she builds internal tooling, works with development teams to develop SLIs and SLOs, and leads the chaos-engineering initiative including coordinating monthly game days.

Kolton Andrus

is the CEO and co-founder of Gremlin. Previously, he managed the resolution of company-wide incidents at Amazon and Netflix.

Dave Rensin

works within engineering leadership at Google. He helps Google deploy its capital to the right geographies, technologies, and strategic investments.

Joyce Lin

is a lead developer advocate with Postman. Postman is used by 7 million developers and more than 300,000 companies to access 130 million APIs every month. She frequently works with API-first organizations, and has strong opinions about do's and don'ts when it comes to modern software practices.

Yury Niño

is a software and DevOps engineer and a chaos-engineer advocate. Niño loves software applications, automating, reading, writing, and teaching.

Paul Osman

is a senior engineering manager at Under Armour. Osman is an experienced software-engineering leader with a passion for operations and reliability engineering.

Jose Esquivel

is a principal software engineer at Backcountry.com.

great tooling to support it—so I founded Gremlin.

Yury Niño: Hi, I am a software engineer from Universidad Nacional in Colombia. Currently, I am working as a DevOps engineer at Aval Digital Labs, an initiative that leads the digital transformation of a group of banks in my country. Also, I am a chaos engineer advocate. I love breaking software applications, designing resilience strategies, running experiments in production, and solving hard performance issues. I am studying how human errors and lack of observability are involved in the safety of software systems. At this moment, I am leading an initiative to create the first chaos community in Colombia. We are building Gaveta, a mobile app for supporting the execution of chaos game days.

Jose Esquivel: Hi, I'm an engineering manager at Backcountry.com where I work within supply-chain management for the fulfillment, marketing, and content teams. They are running about 65 APIs that interact within our network, with other third-party systems, and with public APIs. My responsibility, from a technical perspective, is to make sure these APIs remain stable and that new features are added on time.

Subbu Allamaraju: I'm a VP at Expedia Group. My tenure at Expedia started with bootstrapping a strategic migration of our travel platforms to the cloud. These days, I spend a lot more time on continuing to energize the pace of this transformation, and more importantly, set us up for operational excellence. I feel there is a lot we can learn from each other, so I write on my blog at https://subbu.org and speak at conferences.

Dave Rensin: Howdy! I'm a senior engineering director at Google currently working on strategic projects for our CFO. At Google, I've run customer support, part of SRE, and global network capacity planning.

Paul Osman: I run the Site Reliability Engineering team at Under Armour Connected Fitness. My team works on products like MyFitnessPal, MapMyFitness, and Endomondo. Before joining Under Armour, I worked at PagerDuty, SoundCloud, 500px, and Mozilla. Most of my career, I've straddled the worlds of ops and software development, so naturally found myself gravitating towards SRE work. I've practiced chaos engineering at several companies over the years.

InfoQ: How has chaos engineering evolved over the last year? And what about the adoption—is chaos engineering mainstream yet?

Yee: I don't think chaos engineering itself has really evolved—the concepts are still the same, but our understanding of it has grown. Engineers are starting to learn that it's a principled practice of experimentation and information sharing, not the folklore of completely random attacks. As the myth gets busted and best practices are established, it has fueled adoption. It's not quite mainstream yet, but we're certainly past the "innovator" stage and well into the "early adopter" segment of the adoption curve.

Dickey: Incident response and mitigation is becoming increasingly important and prioritized within organizations. Applications are becoming more complex, outages are becoming more expensive, and consumers are becoming less tolerant of downtime. Fascinatingly, the trend over the past year or two has been away from root-cause analysis or casting blame, and towards thinking of incidents as a perfect storm. Chaos engineering is the ideal counterpart for this industry paradigm shift since it allows engineers to identify and fix issues that could lead to a cascading failure before it ever occurs.

Chaos engineering is mainstream within large tech companies and early adopters, but is still finding its footing within small and medium-sized companies. However, it's a trend that's here to stay, since software isn't getting any less distributed, and humans aren't getting any less error-prone! I'm excited to see what the adoption of chaos engineering and related practices will look like in a few years—it's a fun time to be a part of this industry.

Lin: I believe that chaos engineering is still the new kid on the block that most teams think about implementing one day. Once a team has covered all the bases when it comes to pre-release testing, then initiating chaos experiments becomes a lot more feasible.

Ross: One big thing I've seen come out of the chaos-engineering movement is the popularization of game days. They're not a totally new concept, but I've seen and heard of more and more companies performing them. People are seeing they already have these staging environments where they test software before release, so why not test their incident response process there too? Large companies have been doing this for years, but smaller shops are doing this more and more.

Andrus: Only a few years ago, I often had to explain why chaos engineering was necessary. Now, most teams doing serious operations understand the value and just need some help getting started. The early adopters that have been doing this for a couple of years now are having a lot of success, and are tackling how to scale it across their organizations. The early majority are still testing it out with projects and teams before embracing it more holistically.

Niño: I think chaos engineering has been moving ahead at a staggering speed in a very short amount of time. In 2016, the authors of the classic article "Chaos Engineering" wrote asking if it could be possible to build a set of tools for automating failure injection that was reusable across organizations. Nowadays, just three years later, we have more than 20 tools to implement chaos engineering and several organizations are using them to build more reliable systems in production.

According to the latest Technology Radar published by Thoughtworks, chaos engineering has moved from a much-talked-about idea to an accepted and mainstream approach to improving and assuring distributed system resilience.

Esquivel: Online business has been growing in regard to load, and for e-commerce the majority of the traffic is focused on a few weeks of the year. This means that during this period of high traffic, the stability is directly proportional to the revenue of the company. Adoption of test harnesses like unit, integration, security, load, and finally chaos testing has become mission critical. Adoption has been improving, and we are sure that we need chaos engineering. Now we need to find the time investment to mature it.

Allamaraju: Not yet. Though this topic has been around for about a decade, the understanding that led to this idea called chaos engineering is still nascent in the industry.

Here is why I think so. Very few of us in this industry see the software we operate in our production environments as stochastic and non-linear systems, and we bring a number of assumptions into our systems with each change we make. Consequently, when teams pick up chaos engineering, they start testing operating systems and network and application-level failures through various chaos-engineering tools, but don't go far enough to learn about safety.

Even though most of the work we do to our production systems seem benign, our work can sometimes push our systems to unsafe zones, thus preventing us from delivering the value that customers need and want. So, learning from your incidents before you go deep into chaos engineering techniques is very important.

Rensin: I think chaos engineering has always been mainstream; we just called it something different. If your customers find it before you do, we call that "customer support." If your telco (or provider) finds it before you do, we call that "bad luck." That, I think, is the biggest change over the last year. As more people discover what chaos engineering is, the more they're discovering that they've already been doing it (or needed to do it) in their human systems.

Osman: Interest is exploding, but lots of companies are still struggling with how to get started. I think that's because it's not a magic bullet—chaos engineering exists within a matrix of capabilities that together can be really effective, but without things like observability, a good incident response process, blameless postmortems, etc., you won't likely get results.

Basically, if you're not a learning organization, there's no point in doing experiments, so chaos engineering won't help you. On the bright side, more and more people are realizing that modern systems are complex and we can't predict failures. Just having QA teams test in a staging environment isn't going to cut it anymore, hence the interest in chaos engineering. I think this all probably means it's mainstream now.

InfoQ: What is the biggest hurdle to adopting chaos-engineering practices?

Yee: The biggest hurdle, as with most technological change, is with people. Getting approval from leaders to intentionally break your production systems is a very hard sell, especially without a clear understanding of the business risks and benefits. Creating a solid strategy to manage the blast radius of chaos tests, ensuring discovered weak points are improved, and communicating the business value are all key to winning support.

Dickey: The cultural shift—getting engineers to prioritize and value destructive testing when they have plenty of work sitting in their backlog—has been the most challenging part of driving chaos-engineering adoption at Mailchimp. Planning a game day or implementing chaos-engineering automation requires several hours of pre-work at a minimum, and without a dedicated chaos-engineering team, taking that time to plan and scope scenarios can feel contrary to forward momentum.

So it's important to remember that every engineering norm required a cultural shift—whether that was adoption of unit tests, code reviews, daily standups, or work-from-home options. Since Mailchimp's chaos-engineering program was largely bootstrapped by the SRE team, we had to rely on tech talks, internal newsletters, reaching out to teams directly, and making game days fun (we recommend chaos cupcakes) to drive engagement throughout the engineering department.

Lin: A common pitfall is not clearly communicating the reasons why you're adopting chaos-engineering practices in the first place. When the rest of the organization doesn't yet understand or believe in the benefit, there is going to be fear and confusion. Worse yet, there is often the perception that you're just breaking stuff randomly and without legitimate hypotheses.

Ross: You need a process pusher. It's easy to keep doing what you're doing because that's what you've always done. You need a champion for trying out chaos engineering. In a way I think the term "chaos engineering" might scare some people into trying it, so you need an individual (or a few) to really explain the value to the rest of the team and get everyone onboard. Getting buy-in can be hard.

Andrus: I think many people understand the general concept but underestimate the value it can provide to their business. They see it as yet another thing to do instead of a way to save them time with their ongoing projects, whether it be migrating to the cloud or adopting a new service like Kubernetes. Too often, reliability is an afterthought, so we are putting a lot of work into educating the market and explaining the value of putting in the upfront effort.

Niño: I have had to jump many hurdles. Understanding how to build resilient systems is about software, infrastructure, networks, data, and even about getting the bullet item. However, I think the biggest challenge is the cultural change.

Getting the approval of the customers to experiment with their systems is difficult, in my experience. When we talk about chaos engineering, customers get excited at first sight. For example, we hear expressions such as "if the top companies are doing this, we should look at chaos engineering too." However, when they understand the fundamental concepts, they say things such as "we are not going to inject failure in our production."

On the other hand, I think it's important to mention the limitations also; to clarify that chaos engineering on its own won't make the system more robust—that is a mission for people who are designing the systems. Chaos engineering allows us to know how the system reacts to turbulent conditions in production and builds confidence in its reliability capacity, but designing strategies is our responsibility. The discipline is not a magic box that generates solutions to our weaknesses.

David Woods has a good point on this matter: "Expanding a system's ability to handle some additional perturbations increases the system's vulnerability in other ways to other kinds of events. This is a fundamental trade-off for complex adaptive systems."

Esquivel: Enterprises must be IT savvy so that they can overcome the biggest hurdle in adopting chaos engineering, which is the lack of a culture that gives high priority to stability. Upper management must understand technology and know that investing in quality translates into delivering effective long-term solutions. Delivering functionality is not enough, as whatever is built as a functional feature must guarantee that the following non-functional features are part of the solution: maintainability and consistency through unit and integration testing, performance through load testing, and stability (for example by injecting chaos testing).

Allamaraju: The biggest hurdle, in my opinion, is understanding the rationale behind chaos-engineering practices, developing a culture of operating our systems safely, and articulating the relative value of such practices against all other work. You can't succeed with chaos engineering if you can't articulate and create hypothesis to demonstrate the value it can bring.

Once you cross those hurdles, the actual practice of chaos engineering falls into place naturally.

Rensin: Letting go of perfect. Too many people think that the point of systems is to eliminate mistakes. Once they get the idea that a great system is resilient, not perfect, then chaos engineering comes naturally.

Osman: I think trust is the most common hurdle. You have to have a culture where teams trust each other and non-engineering stakeholders trust that engineering teams have the customer's best interest in mind. That trust is extremely important—if teams fear what might happen if they run a chaos experiment and it doesn't go well, they'll be understandably hesitant.

Chaos engineering reveals one of those inconvenient truths about complex systems: that we aren't fully in control, and can't predict what will happen in production. Unfortunately, some organizations will be reluctant to face this reality. This is why it's incredibly important to have practices like blameless postmortems and a good incident response process in place before you look at introducing chaos engineering. Those practices impact the culture of an organization and make discussions about chaos engineering possible.

InfoQ: How important are the people aspects with chaos engineering? Can you provide any references for leaders looking to improve resilience across an organization?

Dickey: Chaos-engineering adoption is as much a marketing effort as it is a technical one. Leadership buy-in is crucial for a chaos-engineering program to succeed long term. Awesome Chaos Engineering is a comprehensive list of resources that I've found to be a helpful starting place.

Lin: Most organizations have an existing fear of delivering software faster, and mitigate that fear with more and more pre-release testing. Once an organization begins trusting their mean time to recovery, there's a willingness to learn more with high-impact chaos experiments.

Ross: You might uncover some seriously defective code/infrastructure design when you chaos-test—something that might make you say "how did this happen?" It's important to keep in mind that organizations introduce problems, not individuals. You might discover a problem that some code causes and while you won't say their name at all, that person who wrote the code knows it was them. Depending on the person, they might feel bad and maybe even tighten up because of it. You need to make sure that person feels supported and enabled, because we all are going to make mistakes (in fact, if you write code that is in production, you already have). Avoid finger pointing at all costs. If you can't avoid it, you're not ready for chaos engineering.

Andrus: One of my pet peeves in our industry is the lack of investment in training our teams before they are placed on call. There's a reason we run fire drills: it's so we can have the opportunity to practice in advance and build muscle memory so people aren't running around with their hair on fire when something bad happens. Similarly, a large part of incident management is coordination across teams—often teams that haven't interacted previously. Creating a space for those relationships to be built outside of urgent situations allows for greater preparation and better results.

Niño: I think chaos engineering is a discipline created by people, for people. We, the humans, our attitudes, and the trade-offs we are making under pressure are all contributors to how the system runs and behaves. In chaos engineering, we design failures, run experiments in production, analyze the results, and generate resilience strategies.

Regarding references, I think some awesome people and organizations have improved their resilience using the benefits of chaos engineering. The Gremlin team is doing a great job leading this space; the tool and the documentation are very good references for the leaders who are worried about the resilience.

Esquivel: Once people understand that what you build, you own (and by ownership, I mean company ownership) it is easy to gain support, and people will easily participate. At Backcountry, improving resiliency is championed by the VP of product, the director of engineering, and several engineering managers. In this case, the top-to-bottom support has been critical in order to make progress.

Allamaraju: People are at the heart of this discipline. After all, it is people that build and operate our systems. It is people that can listen to feedback from those systems through operational metrics. It is people that can learn from incidents to know when systems work and when they don't. Unfortunately, we're still learning about these through experience, which takes time.

Rensin: People are the most important part of any system. We build and run these things for the people. People are also the biggest barrier (or accelerator). The very best way I can think to create organizational resilience is to periodically "turn off" critical humans. What I mean is "Today, Sally will not be answering email or IM for any reason. Feel free to be social, but she's off limits for the purpose of work." Why would we do that? To find all the tribal knowledge that only Sally has and get it all written down—or at least spread around. Do that with enough people randomly, and suddenly you have a much more resilient team.

Osman: I think, based on my answers above, you won't be surprised when I say that people aspects are critical! The best tools and processes are nothing if people don't trust them and buy into them. In terms of references, it's seven years old now, but I always point people to John Allspaw's article "Fault Injection in Production" that was published in ACM Queue. It's a great introduction to this concept. From the same author and year, the blog post "Blameless PostMortems and a Just Culture" is great. I've mentioned the importance of a good incident response process a few times; if an organization doesn't have a process they're happy with, I always just refer them to PagerDuty's excellent incident response documentation. It's one of those cases where you can just point to it and say "if you don't do anything right now, just do this."

InfoQ: Can you recommend your favorite tooling and practices/games within the chaos space?

Yee: One of the nice things about chaos engineering is that you don't need a lot of additional tools. For example, you can use Linux's native kill command to stop processes and iptables to introduce network connectivity issues. Some other tools we use at Datadog include Comcast, a tool to simulate poor network conditions, and Vegeta, an HTTP load-testing tool.

Dickey: Mailchimp has been a Gremlin customer for about a year, and we've been happy with the flexibility and functionality their tooling offers. We chose to use their tool rather than build our own because we believed that our developers' time was better spent creating amazing products for our customers than developing an in-house tool for testing. We also have some unique technical constraints (come see my conference talk to learn more!) that prevented us from using many open-source tools.

The SRE team runs game days every month, and more often as requested by other teams. We have used incident-simulation game days (sometimes called wargames or fire drills) to help our engineers get more comfortable responding to incidents.

Ross: I really like trying chaos engineering without any tools as a precursor. Basically, you perform a game day where one team of engineers is responsible for breaking the site and not telling the other team what they're going to do (or when—that can be fun). It will reveal all sorts of things about where domain knowledge exists, do people know the process when something breaks, do you have an alert set up for it, do they have the right tools to investigate, etc.

Andrus: The best tooling I can recommend is really getting to know your command-line and observability tools. Be able to grep/sort/cut/uniq a set of logs to narrow down symptoms. Be comfortable with TCPDump or network analysis tools so you can really understand what's happening under the covers. Know how latency distributions, aggregation and percentiles, and what you're measuring are key to understanding your system's behavior.

Also, we recently launched a free version of Gremlin for those just getting started that allows you to run shut-down and CPU experiments.

Niño: For industries that still have many restrictions for deploying to the cloud, I recommend using tools such as Chaos Monkey for Spring Boot. It is a great tool that works very well for conducting experiments that involve latency, controlled exceptions, and killing active applications in on-premise architectures.

I've also been exploring ChaoSlingr, a new tool for doing security chaos engineering. It is built for AWS and lets you to push failures into the system in a way that allows you to identify security issues as well as better understand the infrastructure.

Esquivel: We have gone from built-in tools to commercial software, and our recommendation is to go commercial unless you are willing to invest a lot of time and money. Our weapons of choice against mass instability are Jenkins to kick off load testing using Gatling and Gremlin to unleash chaos under a predefined impact radius that makes us feel comfortable to conduct chaos experiments in production and development environments.

Allamaraju: I don't think there is one favorite tool or practice. Instead, I would look at the architecture of the system, the assumptions we are making, observed weaknesses through incidents, develop hypotheses to test for failures, and then figure out what tools or practices can help test hypotheses.

Rensin: At Google, we run DiRT (disaster and recovery testing) where we physically go into data centers and do terrible things to the machines that service Google's internal systems—like unplug hardware. There's a small team that plans and executes it so that we don't cause customer harm, but mostly our engineers experience it as a real outage. We find a lot of things this way, and the DiRT postmortems are some of the best reading at Google.

Osman: In terms of practices, I really like planning chaos experiments during incident postmortems. When you're going through the timeline of an incident, take note of the surprises—perhaps a database failed and no one really knows why, or maybe a spike in latency in service A caused errors that nobody anticipated—these are all great things to run chaos experiments on so the team can learn more about the behavior of the system under certain kinds of stress. Richard Cook has talked about "systems as imagined" versus "systems as they are found" and postmortems are great opportunities for teasing out some of those differences. The differences are perfect fodder for high-value chaos experiments.

TL;DR

- Software applications are becoming more complex, outages are becoming more expensive, and consumers are becoming less tolerant of downtime.
- Chaos engineering is not quite mainstream yet, but the majority of respondents to this Q&A believe that this is past the innovation stage and well into early adoption on the adoption curve. Incident response and mitigation is becoming increasingly important and prioritized within organizations.
- The core concepts of chaos engineering are well established, but over the past several years the wider communities' understanding of it has grown. Engineers are starting to learn that it's a principled practice of experimentation and information sharing, not the folklore of completely random attacks.
- Software engineers have always tested in production (even if they don't realize it). Chaos engineering can provide a more formal approach.

Sustainable Operations in Complex Systems with Production Excellence

by **Liz Fong-Jones**, Principal Dev Advocate @honeycombio
illustrations by **Emily Griffin** (@emilywithcurls)

[Site reliability engineering (SRE)] and development teams need to be willing to address issues head on and identify points of tension that need resetting. Part of SRE's job is to help maintain production excellence in the face of changing business needs.—Sheerin et al., The Site Reliability Workbook

Production ownership, putting teams who develop components or services on call for those services, is a best practice for ensuring high-quality software and tightening development feedback loops. Done well, ownership can provide engineers with the power and autonomy that feed our natural desire for agency and high-impact work and can deliver a better experience for users.

But in practice, teams often wind up on call without proper training and safeguards for their well-being. Suddenly changing the responsibilities of a team worsens service reliability, demoralizes the team, and creates incentives to shirk responsibilities. The team members will lack critical skills, struggle to keep up, and not have the time to learn what they're missing. The problem is one of responsibility without empowerment. Despite the instinct to spend

money to solve the problem, purchasing more tooling cannot fill the gaps in our teams' capabilities or structures. Tools can only help us automate existing workflows rather than teach new skills or adapt processes to solve current problems.

In order to succeed at production ownership, a team needs a roadmap for developing the necessary skills to run production systems. We don't just need production ownership; we also need production excellence. Production excellence is a teachable set of skills that teams can use to adapt to changing circumstances with confidence. It requires changes to people, culture, and process rather than only tooling.

Silo-ing operations doesn't work

Development teams that outsource their operations instead of practicing production ownership struggle with misalignment of incentives around operational hygiene. Such teams prioritize faster feature development to the detriment of operability. The need for human intervention by operators grows faster than teams can provide. Operationally focused teams, who must keep the system running and scaling up, are left to pick up the mess no matter the human cost in distraction, late nights, and overtime shifts. A service that burns out all the humans assigned to operate it cannot remain functioning for long.

When dev and ops throw code at each other over a wall, they lose the context of both developer and operator needs. Releases require manual testing, provisioning requires human action, failures require manual investigation, and complex outages take longer to resolve due to escalations among multiple teams. Teams wind up either working with a heap of manual, tedious, error-prone actions or develop incorrect automation that exacerbates risk — for instance, they restart processes without examining why they stopped. This kind of repetitive break/fix work, or toil, scales with the size of the service and demands the team's blood, sweat, and tears. In organizations that use a dedicated SRE team structure as well as in organizations with a more traditional sysadmin team, teams can quickly become operationally overloaded.

Over time, operability challenges will also slow down product development, as disconnected knowledge and tooling (between dev and ops) mean that the separate teams cannot easily understand and repair failures in production or even agree on what is more important to fix. Production ownership makes sense: close the feedback loop so that development teams feel the pain of their operability decisions, and bring operational expertise onto the team rather than silo-ing it.

Handing out pagers doesn't work.

Putting development teams on call requires preparation to make them successful. Technology companies split the roles of operations and development decades ago, aiming to find efficiencies through specialization. The modern process of forcing those two functions back together to improve agility cannot happen overnight. The operational tasks that the combined DevOps team now needs to do often feel distinct from the skills and instincts that dev teams are used to. The volume of alerts and manual interventions can be overwhelming, which reinforces the perspective that ops work isn't valuable. And reducing the volume of alerts and manual work requires a blend of both dev and ops skills so many developers resist attempts to introduce production ownership onto a team.

Many commercial tools claim to reduce the inherent friction of DevOps or production ownership but instead further scatter the sources of truth and increase the noise in the system. Unless there is a systematic plan to educate and involve everyone in production, every non-trivial issue will escalate to one of the team's very few experts in production. Even if those experts were inclined to share rather than hoard information, the constant interruptions prevent them from writing thorough documentation. Continuous integration and delivery metrics are a trap if they focus solely on the rate at which software ships, rather than the quality of the software.

Time to detect and repair incidents will remain long if teams cannot understand how the system is failing in production. In order to confidently deploy software, teams must ensure that it does not break upon deployment to production. Otherwise, in an environment with a high change failure rate, teams' hard work gets repeatedly rolled back or fixed forward under stressful conditions. While it is fairer to distribute existing operational pain among everyone on a team, true relief from operational overload can only come from reducing the numbers of pager alerts and break/fix tickets.

Specialization in systems engineering has value and takes time to develop, and historically has been devalued. Product developers don't necessarily have a good framework for deciding what automation to write or bugs to fix for the greatest reduction in noise. Even if there were time to address toil and other forms of operational debt, it also requires planning in advance. An analo-

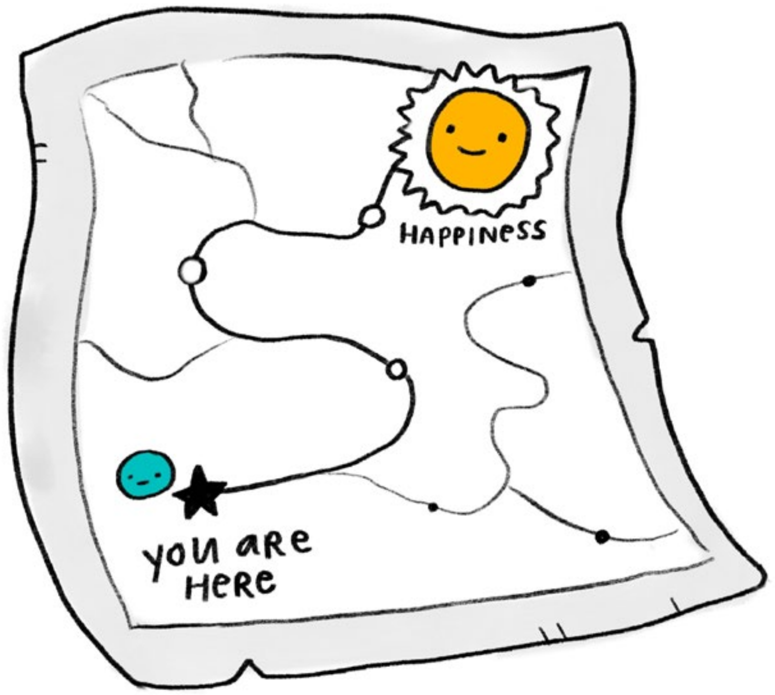
gy is that toil is the operational interest payment on technical debt; paying the interest alone will not reduce the outstanding principal. Production ownership is a better strategy than having a disempowered operations team, but the level of production pain remains the same. Blending team roles certainly spreads the production pain more fairly, but there are better solutions, which reduce that pain instead. Both customers and the development teams benefit when we pay down technical and operational debt.

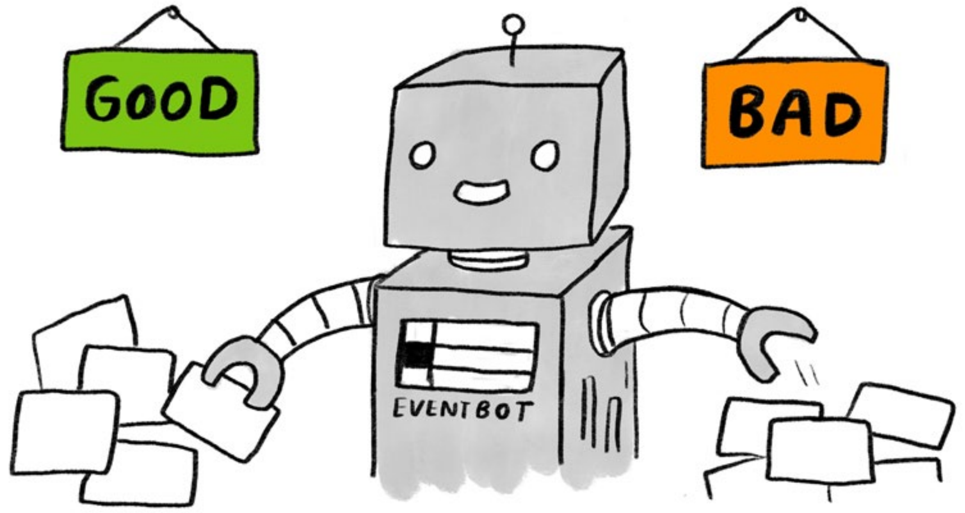
A better approach: Production excellence

Instead of trying to solve the problem of production ownership with tools or forced team integrations, a different people-centric approach is required: that of production excellence.

Production excellence is a set of skills and practices that allow teams to be confident in their ownership of production. Production-excellence skills are often found among SRE teams or individuals with the SRE title, but it ought not be solely their domain. Closing the feedback loop on production ownership requires us to spread these skills across everyone on our teams. Under production ownership, operations become everyone's responsibility rather than "someone else's problem." Every team member needs to have a basic fluency in operations and production excellence even if it's not their full-time focus. And teams need support when cultivating those skills and need to feel rewarded for them.

There are four key elements to making a team and the service it supports perform predictably in the long term. First, teams must agree on what events improve user satisfaction and eliminate extraneous alerts for what does not. Second, they must improve their ability to explore production health, starting with symptoms of user painrather than potential-cause-based exploration. Third, they must ensure that they collaborate and share knowledge with each other and that they can train new members. Finally, they need a framework for prioritizing which risks to remediate so that the system is reliable enough for customer needs can run sustainably.





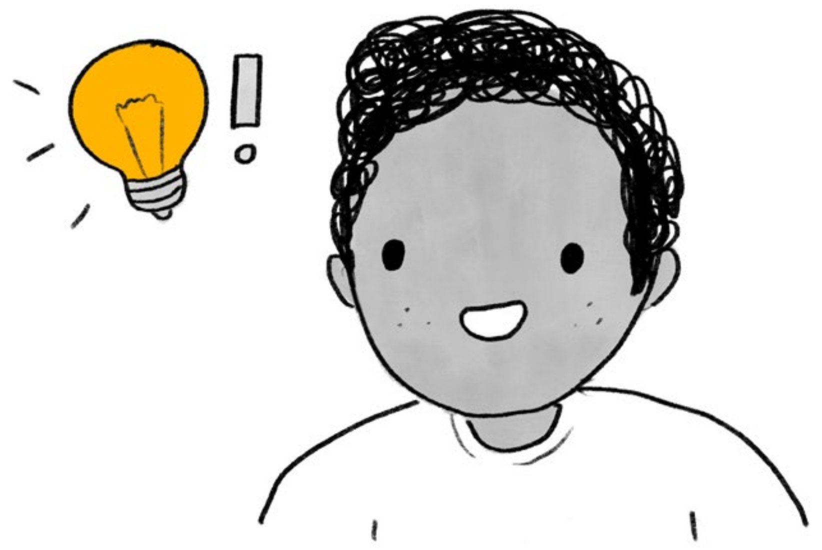
1. Measure what matters

Making on-call periods tolerable starts with reducing noise and improving correlation between alerts and real user's problems. Service-level objectives (SLOs), a cornerstone of SRE practice, help create a feedback loop for keeping systems working well enough to meet long-term user expectations. The core idea of SLOs is that failures are normal and that we need to define an acceptable level of failure instead of wasting development agility in pursuit of perfection. As Charity Majors says, "Your system exists in a continuous state of partial degradation. Right now. There are so many, many things wrong and broken that you don't know about. Yet." Instead of sending alerts for the unactionable noise in each system, we should address broader patterns of failure that risk user unhappiness.

Therefore, we need to quantify each user's expectations through the quality of experience delivered each time they interact with our systems. And we need to measure in aggregate how satisfied all our users are. We can begin to draft such a service-level indicator (SLI) by measuring latency and error rates, logging factors such as abandonment rates, and obtaining direct feedback such as user interviews.

Ideally, our product managers and customer-success teams will know what workflows users most care about, and what rough thresholds of latency or error rate induce support calls. By transforming those guidelines into machine-categorizable thresholds (e.g., "this interaction is good if it completed within 300 milliseconds and was served with an HTTP 200 code," or "this record of data is sufficiently fresh if updated in the past 24 hours"), it becomes possible to analyze real-time and historical performance of the system as a whole. For example, we might set a target for our system of 99.9% of eligible events being successful over a three-month period: an error budget of one in 1000 events within each period.

By setting targets for performance and establishing a budget for expected errors over spans of time, it becomes possible to adjust systems to ignore small self-resolving blips and to send



alerts only if the system significantly exceeds nominal bounds. If and only if it appears that the ongoing rate of errors will push us over the error budget in the next few hours will the system page a human. And we can experiment with our engineering priorities to see if there's spare tolerance and more speed available or we can refocus on reliability work if we're exceeding acceptable levels of failure. If a team chronically falls short of the error budget and users are complaining, perhaps the team needs to prioritize infrastructural fixes rather than new features.

A written team commitment to defend the SLO if it is in danger provides institutional support for reliability work. A SLO that strikes the right balance should leave a user shrugging off a rare error and retrying later rather than calling support and cancelling their contract because the service is constantly down.

SLOs can shift as user expectations change and as different tradeoffs between velocity and reliability become possible. But it is far better to focus on any user-driven SLO, no matter how crude, than to have far too many metrics devoid of correlation with user impact deafen us.

2. Debug with observability

Having eliminated lower-level alerts, what should we do if our system determines that our service is not functioning according to its SLOs? We need the ability to debug and drill down to understand which subsets of traffic are experiencing or causing problems. Hypothesize about how to mitigate and resolve them, then carry out the experiment and verify whether things have returned to normal.

The ability to debug is closely related to the idea of observability, of having our system produce sufficient telemetry to allow us to understand its internal state without needing to disturb it or modify its code. With sufficient knowledge of our system and its performance, we hope to test hypotheses to explain and resolve the variance in outcomes for users experiencing problems. What's different about the requests now taking 500 milliseconds versus the ones taking 250 milliseconds? Can we close the gap by identifying and resolving the performance problem that causes a subset of requests to take longer?

It is insufficient to merely measure and gather all the data. We must be able to examine the data and its context in new ways and use it to diagnose problems. We need observability because we cannot debug systems from only what we've thought in advance to measure. Complex systems failures often arise from new combinations of causes rather than from a finite set of fixed causes. As a corollary, we often cannot reproduce production failures in smaller-scale staging environments and cannot predict them in advance.

Regardless of our approach to observability, it is important to have sufficient information about how requests flow through our distributed systems. Each place where a request traverses a microservice is a potential failure point that we must monitor. This can take the form of wide events with metadata, distributed traces, metrics, or logs. Each approach makes tradeoffs in terms of tooling support, granularity of data, flexibility, and context. We often need multiple tools that work well together to provide the full set of required capabilities. And problems will stymy even the most skilled problem solver if we cannot follow the trail of clues down to specific instances of failing user workflows.

By recording telemetry with sufficient flexibility and fidelity, we don't need a service to remain broken to analyze its behavior. We can restore service to users as quickly as possible, confident that we can later reproduce the problem and debug. This lets us automatically mitigate entire classes of problems, such as by draining bad regions or reverting bad rollouts, and have a human investigate during normal working hours what went wrong. The system's operability improves when humans don't need to fully debug and resolve each failure in real time.

3. Collaboration and building skills

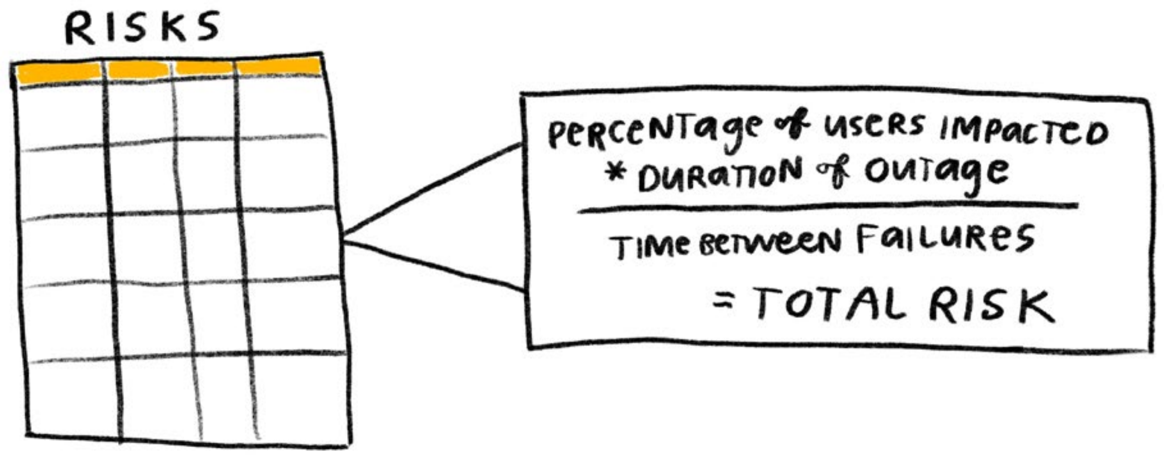
Even a perfect set of SLOs and instrumentation for observability do not necessarily result in a sustainable system. People are required to debug and run systems. Nobody is born knowing how to debug, so every engineer must learn that at some point. As systems and techniques evolve, everyone needs to continually update with new insights.

It's important to develop training programs, to write thorough up-to-date documentation on common starting points for diagnosis, and to hold blameless retrospectives. Service ownership doesn't mean selfishness and silos, it means sharing expertise over time. Above all, teams must foster an atmosphere of psychological safety where team members and those outside the immediate team can ask questions. This empowers individuals to further their understanding and brings new perspectives to bear on problems.

Cross-team interactions can encompass not just upstream/downstream relations but also job roles such as customer support, product development, and SRE. If a customer-support engineer was yelled at the last time they raised a false alarm, they will feel they can't safely escalate problems and the detection and resolution of issues will suffer.

It's critical to the learning process that everyone has some exposure to the way things fail in production. However, many approaches to production ownership are inflexibly dogmatic about putting every developer on call 24 hours a day, seven days in a row. Developers object to participating in processes that are stressful and not tailored to their lives. Fear of having their job description shift under them or of being perceived as shirking duties if they do not comply harms team morale.

Production needs to be owned by engineers of all kinds, rather than just those fitting a specific mold. Involvement in production does not have to be a binary choice of on call or not on call. Involvement in production can take many different forms, such as triaging support tickets for a person who cannot cope with the stress of a pager or being on call only during business hours for those who have family responsibilities at night. Other examples could be a devout Jewish person not being on call during the Sabbath or evening on-call time for a manager who doesn't want pages to interrupt their business-hour one-to-ones. Together, a team can collaborate to fairly share the load of production ownership based on individuals' contexts and needs.



4. Risk analysis

The final element of production excellence is the ability to anticipate and address structural problems that pose a risk to our system's performance. We can identify performance bottlenecks and classes of potential failure before they become crises. Being proactive means knowing how to replace hard dependencies with soft dependencies before they fail. Likewise, we should not wait for users to complain that our system is too slow before optimizing critical paths. We still need a portion of our error budget to deal with novel system failures but that does not excuse us from addressing known risks in the system.

The trick is to identify those risks are the most critical and make the case to fix them. In the 1960s, the American Apollo moon-landing program collated a list of known risksand worked to ensure that these risks cumulatively fell within the program's safety parameters. In the modern era, Google SRE teams espouse the examination and prioritization of risks. They propose a framework of quantitative risk analysis that values each risk by the product of its time to detect/repair, the severity of its impact, and its frequency or likelihood.

We may not always be able to control the frequency of events but we can shorten our response times, soften the severity of impact, or reduce the number of affected users. For instance, we may choose canary deployments or feature flags to reduce the impact of bad changes on users and speed up their reversion if necessary. The success of an improvement is its reduction in bad minutes for the average user. A canary deployment strategy might shrink deployment-related outages from affecting 100% of users for two hours once a month (120 bad minutes per user per month) to affecting 5% of users for 30 minutes once a month (1.5 bad minutes per user per month).

Any single risk that could spend a significant chunk of our error budget is something to swiftly address since it could cause the system to fail catastrophically and push us dramatically over our error budget. We can address the remaining multitude of smaller sources of risk later if we are still exceeding our error budget. And with a quantitative analysis enumerating the fraction of the error budget, each risk will consume, it becomes easier to argue for relative prioritizations and for addressing existential risks instead of developing new features.

Risk analysis of individual risk conditions, however, often fails to find larger themes common to all the risks. Lack of measurement, lack of observability, and lack of collaboration (the other three key aspects of production excellence) represent systematic risks that worsen all other risks by making outages longer and harder to discover.

Production support need not be painful

Successful long-term approaches to production ownership and DevOps require cultural change in the form of production excellence. Teams are more sustainable if they have well-defined measurements of reliability, the capability to debug new problems, a culture that fosters spreading knowledge, and a proactive approach to mitigating risk. While tools can play a part in supporting a reliable system, culture and people are the most important investment.

Without mature observability and collaboration practices, a system will crumble under the weight of technical debt and falter no matter how many people and how much money is ground into the gears of the machine. The leadership of engineering teams must be responsible for creating team structures and technical systems that can sustainably serve user needs and the health of the business. The structures of production ownership and production excellence help modern development teams to succeed. We cannot expect disempowered operations teams to succeed on their own.

TL;DR

- Cultural and process changes, rather than changes in tooling alone, are necessary for teams to sustainably manage services.
- We can reduce pager noise with Service Level Objectives that measure customer experience and alerts for crucial customer-impacting failures.
- Observability, through collecting and querying tracing and events, allows teams to debug and understand their complex systems.
- Teams must be able to safely collaborate and discuss risks to work effectively.
- Services run more smoothly when quantitative risk analysis allows teams to prioritize fixes.

SPONSORED ARTICLE

Gremlin

Incremental Reliability Improvement

Getting to the Next Nine of Availability

If you improved reliability by just 1% each day, how long would it take for you to get that “extra 9?” That’s an interesting question. This article begins by exploring the potential for small improvements to add up like compound interest. It concludes with a list of practical steps we can take that improve system reliability.

Reliability is how well you can trust that a system will remain available. It is typically expressed as availability and quantified using a percentage of uptime in a given year. We care because we often have service level agreements (SLAs) that refer to monthly or yearly downtime in customer contracts. Downtime costs us money. Uptime rewards us with happy customers and decreased expenses (not to mention fewer pager calls and happier Site Reliability Engineers and DevOps teams).

Nines of Uptime and Availability

A site that achieves 99% availability sounds pretty good at first until you realize that 1% downtime equates to 3.65 days of unavailability in a year. Three nines (99.9%) availability computes to 8.77 hours of downtime in a year. For my personal blog, that is fine. For an e-commerce site that relies on customers being able to spend money as they buy things, even that amount of downtime can become expensive quickly.

Today, forward-looking companies strive for high availability (HA), meaning four nines (99.99%, 52.6 minutes of downtime a year) and even five nines (99.999%, 5.26 minutes of downtime) availability. Some even aim for zero downtime, which is vital in something like an aircraft control system, a communications satellite, or the Global Positioning System (GPS) where any downtime could mean human casualties. How does anyone achieve such reliability?

Stanford University’s Dr. B. J. Fogg proposes that we can improve ourselves immensely over time by making small, incremental, easy changes day by day. Change one small thing a day and watch how it adds up, kind of like compound interest. This aggregation of marginal gains pays big dividends if you persist in pursuing them.

If we improve something by 1% daily (compounding daily), it will be 37 times better in a year (1.01365 = 37.78)! Is this real-life feasible? No, we are not really going to achieve exponential change without limits and math-lovers will correctly note that the equation and curve should illustrate logistic rather than exponential growth. But, this simple idea does illustrate how an intentional active pursuit of small improvements can add up to make a big difference over time.

The corollary is as chilling as that initial assertion was encouraging. If we instead allow that same thing to get just 1% worse daily, the degrading also compounds, like this: 0.99365 = 0.03. By the end of that same year, those minor setbacks that are not dealt with degrade things down to just a statistical difference away from nothing. This matters since we want to avoid drifting into failure and entropy is real with any constantly evolving distributed system.

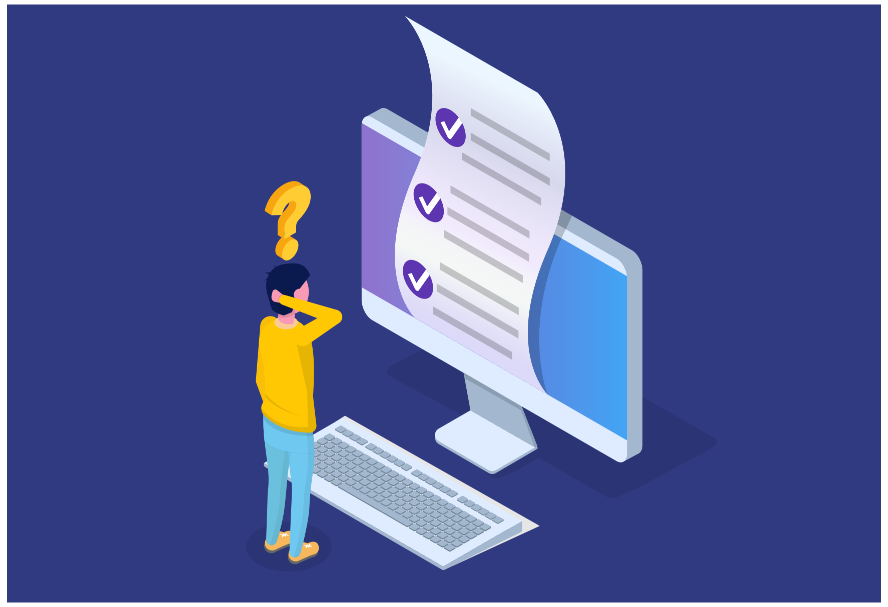
Please read the full-length version of this article here.

Unlocking Continuous Testing: The Four Best Practices Necessary for Success

by **Lubos Parobek**, Product at Sauce labs

Not long after bursting onto the scene as a revolutionary new approach to software delivery, agile development is already at an inflection point. It started with the best intentions, of course. As it became increasingly clear that the success or failure of a business in the digital age was dependent on the ability to rapidly deliver flawless web and mobile applications to customers, organizations began shifting en masse toward new agile development methodologies, believing that doing so would lead to faster release cycles, better product quality, and happier customers. To great fanfare and considerable hype, the age of agile development had arrived.

Yet, with the end of the decade now in plain sight, something unexpected has happened. Agile development has quietly stalled out. In fact, the pace at which organizations release software is actually on the decline. According to Forrester (Forrester: The Path to Autonomous Testing: Augment Human Testers First, Jan. 2019), the percentage of organizations releasing software at least on a monthly basis declined from 36 percent in 2017 to 27 percent in 2018.



In other words, for most organizations, the promise of agile development has failed to materialize.

Testing as the Fulcrum Point

While it wouldn't be accurate to say that any one factor is 100 percent responsible for a trend as significant as the stalling out of agile development, it's also by no means an over-generalization to say that testing is one of the single biggest roadblocks standing between organizations and their agile objectives. In fact, in a recent Gitlab developer survey, testing was identified as the most common source of development delays, cited by more than half of respondents.

While the majority of organizations have enthusiastically embraced agile development, most still find themselves unable to effectively implement continuous testing throughout the software development lifecycle.

And therein sits the roadblock. Organizations are now finding out that when it comes to agile development and continuous testing, you can't have one without the other. If you're unable to test your applications quickly, reliably, and at scale, then you're inevitably left with a choice: slow down your delivery process on account of testing and risk missing your release date, or stick to your release date and risk delivering a poor quality application to your customers. Neither choice is a good one, and both defeat the purpose of implementing agile development in the first place. Agile development should be about **not** having to choose between speed and quality. (*Accelerate* by Nicole Forsgren, Jez Humble and, Gene Kim is a great book for anyone who wants to dig deeper into this concept.)

Overcoming the continuous testing roadblock is thus the key to delivering quality apps at speed and fully realizing the promise of agile development. Here are four critical best practices organizations can implement to do just that.

1: Focus on Test Quality

If you're serious about succeeding with continuous testing, test quality is everything. That's not to say there aren't other important considerations (the most important of which are addressed later on in this article), but just about every other continuous testing best practice is only relevant to the extent that you're starting with a foundation of high-quality tests. And the most direct and reliable determinant of test quality is pass rate.

You should be writing, managing, and executing test suites so the overwhelming majority of your tests pass. Now, *should* every test pass? Absolutely not. No developer is perfect, and a small percentage of tests should fail. The entire reason we test applications is to discover bugs and fix them before they're pushed into production and create a poor user experience, so a test that exposes a potential flaw in your application is a test that has served its purpose well. That said, there's a considerable difference between tests failing on occasion, and tests failing with regularity.

When tests fail on occasion, developers can safely assume the new code they're testing has caused something in the application to break, and they can quickly take action to remedy the problem (assuming, of course, that they're following the next best practice as well). But when tests are consistently failing, developers begin to rightfully question the results, uncertain as to whether the failure was caused by the newly introduced code, or whether it is instead reflective of a problem with the test script itself. When that happens, manual follow-up and exploration are required, and the agile development process into which you've invested so much time and energy screeches to a halt.

Having worked directly with many QA teams, including large enterprises running millions of tests each year, I generally advise that organizations should aim to pass at least 90 percent of all tests they run. In my experience, that's usually the breaking point at which the number of failed tests starts to exceed an organization's ability to manually follow-up on those failures. Thus, organizations that find themselves consistently below this threshold should place greater emphasis on designing and maintaining test suites in a manner that will lead to better a pass rate, and allow them to avoid scenarios where the number of failed tests exceeds their bandwidth to implement the appropriate manual follow up. The best way to do that is through our next best practice.

2: Keep Your Tests Short and Atomic

One of the most reliable predictors of test quality is test execution time. It makes sense: the longer and more complex a test becomes, the more opportunity there is for something to go wrong. The shorter a test is, the more likely it is to pass. In fact, according to a new industry benchmark report based on millions of actual end-user tests, those which complete in two minutes or less are two times more likely to pass than those lasting longer than two minutes.

Suites with shorter tests are not only more stable, but they execute faster as well. Remember, agile development is first and foremost about speed. The faster your tests execute, the faster you can get feedback to developers, and the quicker you can deliver apps to production and get that new release into the hands of your customers. Now, it might seem obvious that shorter tests execute faster than longer tests, but most organizations focus on the number of tests within a test suite rather than the length of those tests, mistakenly assuming that a suite with just a few long tests will execute faster than a suite with many short tests. If you're running tests in parallel (forthcoming best practice spoiler alert), the suite featuring many short tests will actually execute exponentially faster than the suite with just a few long tests.

Take a sample scenario where you have one test suite featuring 18 long-flow, end-to-end tests, and a second suite featuring 180 atomic tests. In almost every instance, the suite featuring 180 atomic tests will executive significantly faster than the suite featuring just 18 tests. In fact, when we model this exact scenario for customers during live demos, the suite featuring 180 atomic tests typically executes 8 times faster than the suite with 18 long-flow tests.

The Power of Atomic Tests

If you want to keep your tests short—and you should—the best way to do so is by keeping them atomic. An atomic test is one that is scripted to validate just one single application feature, and nothing more. So, instead of a single test to validate that your home page loads, visitors can log in with their username and password, items can be added to a cart, and a transaction can be successfully processed, you would design four or five separate tests, each measuring just one of those aforementioned pieces of functionality.

Atomic tests are also considerably easier to debug if and when a test does fail. Being able to quickly get feedback to your developers is great. So is having complete confidence that a failed test signals a break in the application rather than a flawed test script. What is most important is the ability to quickly fix what's broken, and atomic tests make it far easier for developers to do just that.

For starters, because atomic tests are inherently short and thus execute quicker than longer tests (see the sample scenario above), developers are usually receiving feedback on code they only recently wrote. Fixing code you wrote just a few minutes ago is considerably easier than fixing code you wrote hours or days ago. In addition, because atomic tests focus on just a single piece of application functionality, when one does fail, there's generally no confusion about what's gone wrong. After all, it can only be that one thing. Developers thus don't have to spend precious time and energy trying to diagnose the root cause of the problem. They can immediately remedy the bug and quickly get back to the world of developing great software.

3: Test across Multiple Platforms

So, you're scripting short, atomic tests, achieving a high pass rate, and quickly remedying bugs as soon as they're identified. You're home free, right?

Not quite. Customers in today's digital world consume information and services across an ever-growing range of browsers, operating systems, and devices. To truly realize the promise of agile development, organizations must rapidly deliver high-quality applications that work as intended whenever, wherever, and however customers want to access them.

If a customer wants to access your website from a PC using a slightly older version of Firefox, that website needs to look great and function perfectly. If a different customer wants to access your site from an iPad using the latest version of Chrome, the site needs to look great and function perfectly. And if a third customer wants to access your native mobile app from an Android phone, you guessed it, the app needs to look great and function perfectly.

The ability to quickly determine whether an application functions correctly across an ever-growing range of device, operating system, and browser combinations is a critical component of effective continuous testing. This includes both mobile and web browsers and operating systems, as well as real devices. Once again, based on my experience working with hundreds of enterprise customers to help them execute millions of tests, I recommend striving to test across at least 5 platforms (defined as any combination of a desktop or mobile operating system and browser) with each test implementation. This will usually give you the breadth of coverage you need to confidently release your app on the platforms your customers are most likely using.

You should also strive to incorporate real-device testing into your overall continuous testing strategy as well. Doing the latter usually requires an organization-wide shift to a "mobile-first" (or at least, "mobile equal") mindset, in which a proportional amount of time and resources are dedicated to ensuring that updates to mobile web and mobile native applications keep pace with updates to web applications.

4: Leverage Parallelization

Even if you're brilliantly executing each of the preceding best practices, you simply cannot scale to meet the needs of your growing digital enterprise without parallel test execution. Without parallelization, test suites will eventually take too long to run, and your automated testing initiatives will invariably fail.

To understand why parallelization is so important, consider the hypothetical example of a test suite with 200 (hopefully atomic!) tests, each of which takes 2 minutes to complete. If you can run those 200 tests in parallel, you can execute the entire suite in just two minutes, giving your developers immediate access to insight on the validity of at least 200 application functions. If you had to run those same 200 tests sequentially, however, it would take nearly 7 hours for you to get that same amount of feedback. That's a long time for your developers to be waiting around. (And when your developers have to wait around, your customers inevitably do too.)

The ability to execute tests in parallel is thus table stakes for effective continuous testing. The good news? If you're following the best practices I've already outlined, you're well on your way. Running tests that are atomic and autonomous (that is, that can execute completely independent of any other tests) is the most important step you can take to position yourself to effectively leverage parallelization. Beyond that, pay attention to design your test environment so that you have enough available capacity (usually in the form of VMs) to run tests concurrently. Just as importantly, leverage that capacity to the fullest extent possible when executing your test suites.

Success or Disillusionment?

Theodore Roosevelt is often credited with the saying that "nothing worth having comes easy." More than 100 years later, those words hold true for continuous testing. Continuous testing is not easy. But there is a roadmap for success, and it revolves around the key pillars outlined in this article. To recap:

- Shorten test execution time by running test suites with many small tests rather than those with just a few long ones
- Ensure test quality by always running atomic tests
- Ensure proper test coverage by expanding the number of platforms against which you test
- Run tests in parallel to ensure scalability never becomes a roadblock

I'd like to close by revisiting the inflection point at which agile development teams now find themselves. There are two possible paths forward. One is marked by success, one by disillusionment, with little gray area in between. If you haven't already, the time is now to commit to achieving continuous testing excellence, and finally turn the promise of agile development into a lasting reality.

TL;DR

- Organizations have shifted en masse toward new agile development methodologies, believing that doing so would lead to faster release cycles, better product quality, and happier customers.
- However problems are afoot: According to Forrester, the percentage of organizations releasing software at least on a monthly basis declined from 36 percent in 2017 to 27 percent in 2018.
- Testing can be one of the single biggest roadblocks standing between organizations and their agile objectives.
- While the majority of organizations have enthusiastically embraced agile planning and development, most still find themselves unable to effectively implement continuous testing throughout the software development lifecycle.
- There are four best practices to help overcome this: focus on test quality, keep your tests short and atomic, test across multiple platforms, and leverage parallelization.

Testing in Production—Quality Software, Faster

by **Michael Bryzek,** CTO and co-founder of Gilt Groupe
Presentation summary by **Manuel Pais**

Testing in production gives us a great tool to be able to accelerate our software development and produce higher quality software faster.

Think about the last feature that you worked on and deployed to production. You went through your software development and code review process, code reviewed it, felt great, and went home. Is that feature working in production right now? How do you know?

Maybe your inbox isn't getting spammed with alerts from Nagios or Graphite. But is that enough? How can you be totally sure that your feature is actually working, not just at the time of deployment and verification, but a week later, a month later, a year later? If you can know that, then you can reduce some of the anxiety and focus your energy full-time on what's ahead.

The dreaded outcome is that ultimately, we hear from customer service. The phone rings, and we realize that we missed something. It's not a great feeling. Ultimately, testing in production is about removing that anxiety and knowing that the software we've built is working.

Software Quality is Hard

A few years ago, an article was published about the people who write software for the space shuttle, and it's amazing. People work 9 to 5. They're steady. When they find a bug, they fix the bug and review why it happened. Then they look at other places in the entire code base where conditions exist for a similar bug to happen, and they fix them all.

When they're preparing for a software release, they use data to assess the quality of the software. If every new release has 120 bugs, on average, and they've only found 50 so far in the current version, they don't release it until they find more bugs—the unknowns.

They produce some of the highest quality software in the world, but it comes at a cost. This is the most expensive software ever produced in the world.

A few years ago, salesforce.com made a huge investment in testing their cloud software and scaling all of their integration tests across their APIs. They had approximately 50,000 tests, and if you ran them serially, they'd run for a decade or so. They made a massive investment in tooling and technology to scale.

At Facebook, thousands of engineers work on software to detect changes in the code review process automatically, before they become issues for customers. All this is just a signal that it's hard to ensure software quality.

Delivering Quality Software

When we think about software quality, we tend to think about the software development process up until the point that we push code to production. But, if we really think about the end-to-end lifecycle of code, it's everything from when a feature or idea is conceived, all the way until that software is sunsetted. We want the software to keep a high quality throughout the entire lifecycle.

Production is a really important part of our lives as developers and engineers. Testing in production has a bad name, but it's an incredibly powerful technique to help us build quality software. It allows us to prove that our software works, and interestingly, leads us to test software in a number of different ways.

At Flow, we practice true continuous delivery. That means when you merge a pull request, that code is going to production—there's nothing else standing in the way. Continuous integration happens before the commit is merged. It's an important distinction because we wanted to make sure that the process used to move code to production is the same in times when there's no stress and in urgent times when a bug needs to be fixed quickly. The pipeline is exactly the same.

We don't have any staging environments at Flow. We don't have any developer environments. We don't have any QA environments. We don't have any pre-production environments.

Now, if you're a developer, what do you want? You want your code to be in production, without causing any issues. If you're uncomfortable pushing the deploy button, write some tests. If you're still uncomfortable, write some more tests. Eventually, you'll write enough tests that you feel comfortable your code can go to production. Now we can talk about which portion of those tests are safe to continue to run in production and how we can do that.

True Continuous Delivery

You can't understand the full benefits of continuous delivery until you've truly practiced it. When you merge code and it goes to production safely and predictably every time, this is a fantastic way to operate. When your deploys are really that simple and reliable, it opens up new ways of producing code.

If we want our engineering groups really invested in writing good automated tests, we need to remove the safeguards. If the only way we know that our software is going to work in production is with a good automated test, then we're motivated to write it.

Also, when releasing software is simple and predictable, we can make small releases all the time. That also means we can release a configuration change in exactly the same way as code, with the same visibility. The monitoring systems that flag deploys in production will do the same for configuration. When checking what happened in a deploy, we will see that a configuration state changed.

At Flow, we wrote a small open-source continuous delivery tool called Delta. Delta listens to webhooks from GitHub, builds Docker images on Travis, and scales clusters on AWS ECS and inspects their health when necessary. When the tool detects a change on master, it creates a new tag and safely deploys the change to production. It's an extremely lightweight tool. The key point here is that having the continuous delivery process in place was the hard thing—not the tooling.

No Staging Environments

"I love my staging environment," said nobody ever.

Staging environments feel right for deploying changes and visually inspecting them, or to have another team run through test scenarios. But as we move into microservice architectures, a few things start to happen. First, the staging environment starts to change during the testing. We can't serialize changes for verification, otherwise we won't be able to change much in a day. Such bottlenecks would be inefficient and slow us down.

Secondly, it's very difficult to treat a staging environment as a production environment. If your tests depend on a service out of your control, it could be blocked due to a number of reasons: that service is unavailable, or unresponsive, or incorrectly configured, or missing the right test data, or out of memory. Even if there are alerts for these situations, they likely won't be treated by the team responsible for that service in the same way as a production issue. Therefore, staging becomes unreliable and we don't have the means to fix it.

Many organizations start off with a single staging environment, and when they move into a distributed architecture they branch off and start creating a staging environment for each team/service (for example, the payments team staging environment, the checkout team staging environment, the login staging environment, and so on). What ends up happening is that they have disparate failures—sometimes they work, sometimes they don't.

Teams typically are under pressure to release their features. So, they start hacking their staging environment until it's consistent enough that they can test the feature they need to release. This gives them some assurance that the feature will work in production. But they can't really know until it's live in production.

When a problem occurs, staging environments are fragile and difficult to debug. You have to spend considerable effort to reason with failures in a staging environment. For example, I once wrote a small Ruby script to find all the log files in the system. Every three seconds show me the top five most changed longs as those were probably due to errors.

Organizations where I've seen staging environments work, invest more than half their budget in staging. Under that number, most organizations have what I call dysfunctional staging environments which require large efforts across the organization to make it work.

Finally, staging environments create the wrong incentives by asking people to repeat things over and over again, in most cases without automation. If you can release and inspect into a staging environment, the incentive to write the automation decreases, because it's hard to write good automated tests that are repeatable.

Learn to Trust Your Tests

If you remove staging, and you still want to have good production quality code go out, then the question really boils down to your tests. If you're not comfortable that your code is going to work, write more tests.

With that level of trust in our tests, we can then apply practices, such as a "dependents day." At Flow, we decided that all of our code will depend on the latest versions of everything: our own code, our own libraries, the open-source libraries that we use, AWS, etc. During dependents day we automatically find the latest versions of everything that we use and submit pull requests to all the services to update dependencies. If the resulting build is green, we merge and deploy that change to production because we have built enough confidence in our tests.

Now we don't have to worry about version mismatches or upgrades for services running against old versions for weeks, months, or even years.

Quality through Architecture

There are a number of elements to achieve quality through architecture. One of them is the idea of extreme isolation. If your service is not functioning because something else broke it you will wonder "How can I run a reliable service if things out of my control can break my service?"

One approach is extreme isolation of your service. For example, services at Flow have their own DNS, load balancer, and private database. No other service is allowed to connect to the database and there is no shared state or console. This stops cascading failures and prevents external services from causing an issue in your service.

Engineers might consider this costly, but the cost of understanding failure is much higher than the additional cost of running the extra load balancers or infrastructure. The benefits of being able to regain trust in our services quickly and drive quality are much higher than the cost.

This approach means that our network of microservices is very quiet. Services don't talk directly to each other—they consume data through rich event streams containing relevant messages published by other services. There's only a handful of use cases that really require the services to be synchronous. For example, if you submit an order, you want to make sure that you have received payment for the order. But those are the minority and everything else is asynchronous.

With event streaming, if a service we depend on has an issue, the effect on our service is a slight delay in receiving the data through the event streams, and that's much easier to manage than cascading failures.

Example: Know That Checkout Works

Let's look at a few real examples of successfully testing in production.

At Gilt, an ecommerce company, we wanted to be sure the functionality to submit an order is working at all times. The common approach is to test in staging, then deploy to production and maybe set up an alert that goes off if no order has been placed in the last 5 or 10 minutes.

What if we rethought this and actually submitted an order ourselves (with a bot) every minute or every five minutes? What would have to happen for that to take place?

For us, this means first having a way to identify a test user in production. We can also check that registration and login are working when we create this new test user. We agree to identify test users by a domain name (for example, gilttest.com) and any order from that domain will get canceled. This means only a three line change in our order processing:

```
if user.
emaildomain=gilttest.com
  order.cancel
end
```

This leads us to the next requirement. The cancelled orders are still claiming inventory for a short period of time, and could prevent real users from buying the ordered items. To solve that, we can query the inventory for items with say more than 1,000 units available, and order those in our production tests.

Now we can write an alert that goes off when there's no order from our test user in the last three minutes, for instance. That's a high fidelity alert that we have a problem in production. We'll be able to investigate and fix the problem faster, ideally before any client notices it. In the process, we will learn from this issue and get better at what we're doing.

Load testing in production is very powerful as well. At Gilt, the business model was every day at 9am PST time, a new selection of inventory went on sale at great prices, and everything sold out. We saw 50x traffic increase between 8:59 am and 9am. And in the early days at Gilt, 80% of revenue took place between 9am and 10am. The cost of an error during that time was extremely high.

But what if we could do a load test at 1am PST when traffic is super quiet? If we identify a problem at 1am, we still have eight hours to fix it before crunch time. It feels scary at first and probably things will fail in production (even if they pass in staging). But with this feedback loop of running the load tests in production and fixing them when problems occur, we sharply increase the quality of the overall system. Now failures are rarer, and when they happen we still have a good chance at protecting the revenue for the business.

Example: End-to-End Integration Tests

We have an integration test that runs once a day via TravisCI cron feature. The test creates an order, makes the payment using a credit card token and then captures funds against that credit card's authorization.

We've invested the time (about three weeks elapsed time finding and fixing different kinds of issues from running in production) to ensure this test is reliable (it has been green for a whole year after those first weeks). Now if this test fails, we know immediately that there's a real problem. We know every single day if we can accept orders and process payment. This is extremely powerful. We know that when we deploy changes to our payment system, we can trigger this test and make sure it works against the payment system. We know that when we change API routes, we can run this test and be sure that the heart of the system continues to work. In other words, we know that things are working as expected.

Example: Verifying Proxy Server Works as Expected

At Flow, we wrote our own reverse proxy to allow all of our microservice APIs to have common authentication handling. This is a critical piece of infrastructure for us. If there's an error in our API proxy, all of the APIs become unavailable—it's a cascading failure—an emergency for us.

Earlier, I mentioned extreme isolation in our services. In this case, the API proxy is the only service with no isolation. This API proxy has its own tests, but how do we ensure that a configuration change or a new feature continues to work?

What we do is run tests with the AWS profile configuration from our `flowvault` account from our PCI environment:

```
$ eval $(AWS_
PROFILE=flowvault dev sbt
--env production)
```

dev is a script that handles everything for developers, so the above command effectively retrieves the production environment for the reverse proxy, but then runs it locally.

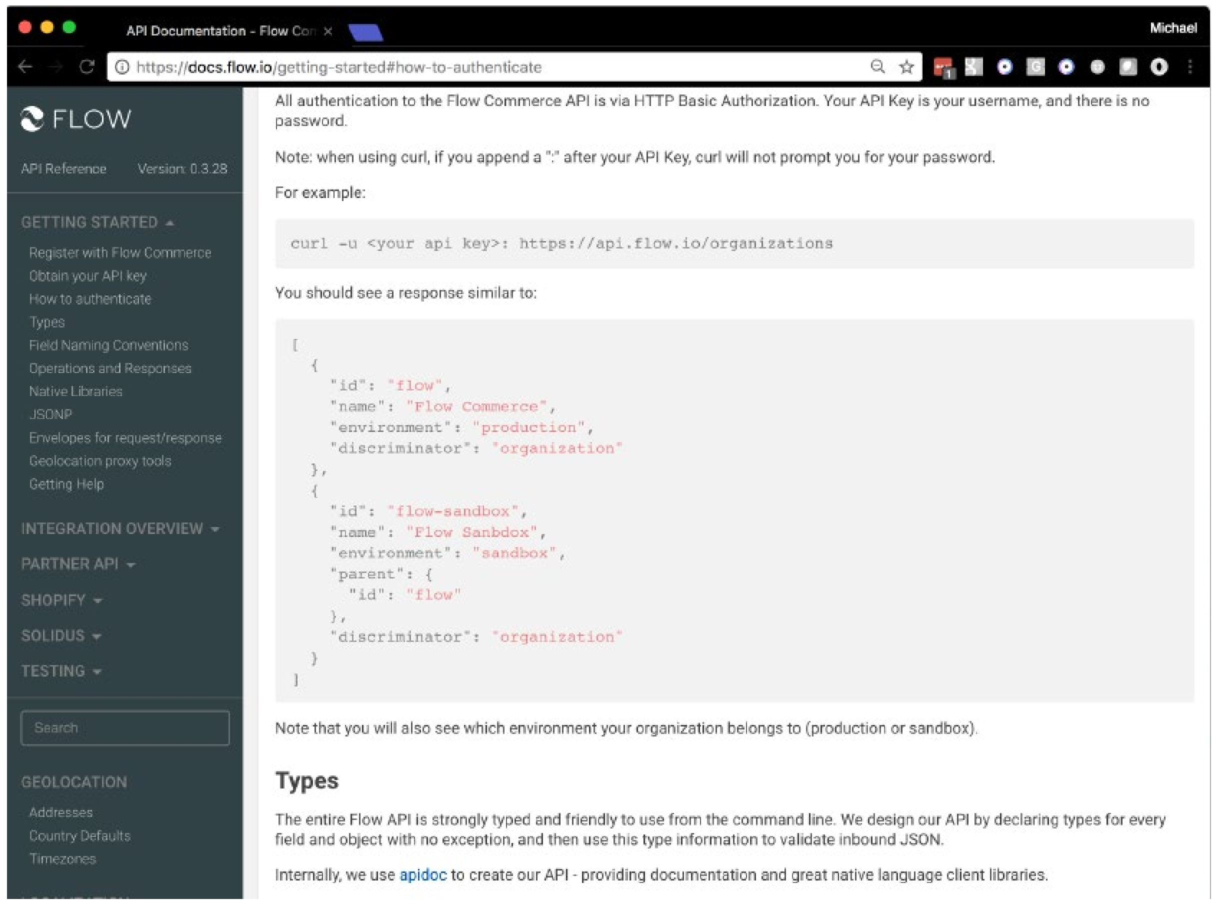
Now I have a reverse API proxy running locally, but talking to our production services. Then I can run tests locally, but instead of hitting the API in production, we're hitting the reverse proxy in localhost. Essentially, for every feature that we add into the reverse proxy, we also add a small test to verify that it works.

Now we can run these tests not only before deploying proxy changes, but also schedule them to run every single day as well.

One other side benefit is that it's actually easier to write HTTP tests for API proxies, that is, inspecting from an external point of view rather than internally. By introducing this way of testing, we were able to greatly simplify the tests that run against the proxy.

What to Do When Things Go Wrong

Let's look at four aspects to consider to reduce the chance that things will go wrong. First of all, make production access explicit, not the default. We need to start thinking about how to test changes in production without access to the production environment at the design phase.



All authentication to the Flow Commerce API is via HTTP Basic Authorization. Your API Key is your username, and there is no password.

Note: when using curl, if you append a ":" after your API Key, curl will not prompt you for your password.

For example:

```
curl -u <your api key>: https://api.flow.io/organizations
```

You should see a response similar to:

```
[
  {
    "id": "flow",
    "name": "Flow Commerce",
    "environment": "production",
    "discriminator": "organization"
  },
  {
    "id": "flow-sandbox",
    "name": "Flow Sanbdox",
    "environment": "sandbox",
    "parent": {
      "id": "flow"
    },
    "discriminator": "organization"
  }
]
```

Note that you will also see which environment your organization belongs to (production or sandbox).

Types

The entire Flow API is strongly typed and friendly to use from the command line. We design our API by declaring types for every field and object with no exception, and then use this type information to validate inbound JSON.

Internally, we use apidoc to create our API - providing documentation and great native language client libraries.

Documentation page with a response example taken from a production test

Second is to use defined paths. If you have an API, use API calls. Don't just hack into a database in production and change some data to enable a specific use case. The point here is to test using the same paths as everything else. However your software works in production, use that. Here is an anecdote from a friend: When his company was approaching peak season, they decided to increase the number of servers and deploy less frequently. Two weeks later, things start crashing. It turns out they had a memory leak for years, but because they were continuously delivering their software, it didn't matter. They changed how they were managing production, and new issues surfaced.

Restricting sensitive data is another important consideration, especially when dealing with compliance standards like HIPAA, PCI, or GDPR in Europe. The earlier example shows that you can run a production test that actually creates a test credit card, verifies the card and authorizes capturing funds. It requires thinking through and fixing unexpected issues but it's worthwhile in the end.

Finally, designing for side effects in advance is important as well. Problems like A/B tests affected by test orders, and not being able to make sense of the result of the A/B test. It's important to think about those possibilities upfront. Figuring out how to cleanse the data created and/or making sure that data is random and normal, thus having negligible impact on the business.

Unexpected Benefits of Testing in Production

Besides the known advantages I've mentioned, there are also some unexpected benefits from testing in production.

One is perfect documentation. We all know static documentation gets outdated quickly. Instead, we can actually capture the request and response as part of our production testing. At the end of the test, if it's successful, it uploads the capture information to S3. We have a tarball with all of our examples. When we release our documentation site, we simply download the latest tarball. We know that the examples are real and working.

The second unexpected benefit is more engaging demos. We are an enterprise software company and one of the things we like to demo is our analytics page. By running the automated tests in production with our demo account, we are also gathering real data that feeds into this analytics demo and makes it much more appealing than before.

Tooling

We do a lot of mocking in our integration tests. We developed a tool called API Builder that can provide a mock of an interface of any service or event stream. The mock knows how to generate valid data and we can override the parts that we need, for example having a geolocation test with specific responses for given locations. Because these mocks are generated from the live services, they respect the same contracts. Therefore, we don't have situations where tests work with the mocks but fail in production.

On the database side, we use a tool called VividCortex which provides real-time feedback on what's happening in our databases. Very quickly, we can see if a given database query is getting slower and slower. The key here is to consider the data side so that if an unexpected problem happens, we know about it as quickly as possible.

Regardless of which tool you use, aggregated logging is critical in order to have a consolidated, fully searchable view of every log file from every service. This allows defining real-time alerts whereby any messages with a given prefix (for instance, "FlowAlertError") trigger a notification email or PagerDuty, for example.

Conclusion

Trust your tests and run them constantly in production (or at least a subset) in order to be sure your software is working as expected. You will need to invest in continuous delivery as well, make it easy to deploy small changes and quickly rollback if necessary.

There are many techniques to help with testing in production, from sandbox accounts to mocks. The end goal of knowing features are working as expected in production (not just at deployment time) is worth investing in those techniques.

TL;DR

- Trust your tests and run a subset in production all the time to be sure the software is working every day, not just at deploy time.
- Invest in continuous delivery to be able to release (and quickly rollback) small changes with high visibility.
- Staging environments, especially in a distributed services architecture, are fragile, difficult to understand, expensive, and create the wrong incentives.
- Architecting and building our systems to reduce dependencies that can cause issues in terms of quality or uptime is an incredibly powerful tool.
- With the safeguard of good automated tests in production, we can instrument production for real-time feedback for unexpected issues.