

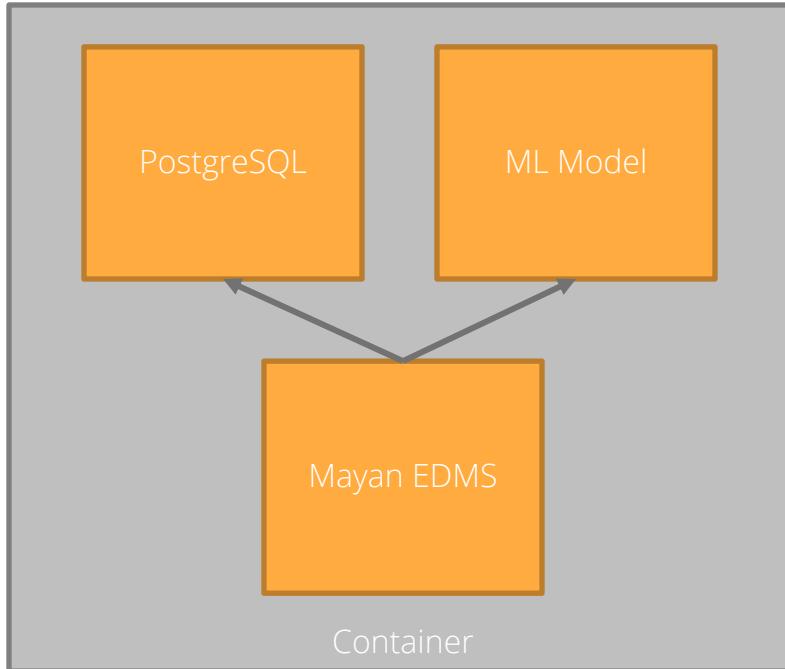
# Chaos Engineering

Building confidence in your application and team  
through failure experimentation

Christopher S. Meiklejohn

Teaching Assistant  
Carnegie Mellon University

# Exercise: Monolithic Application



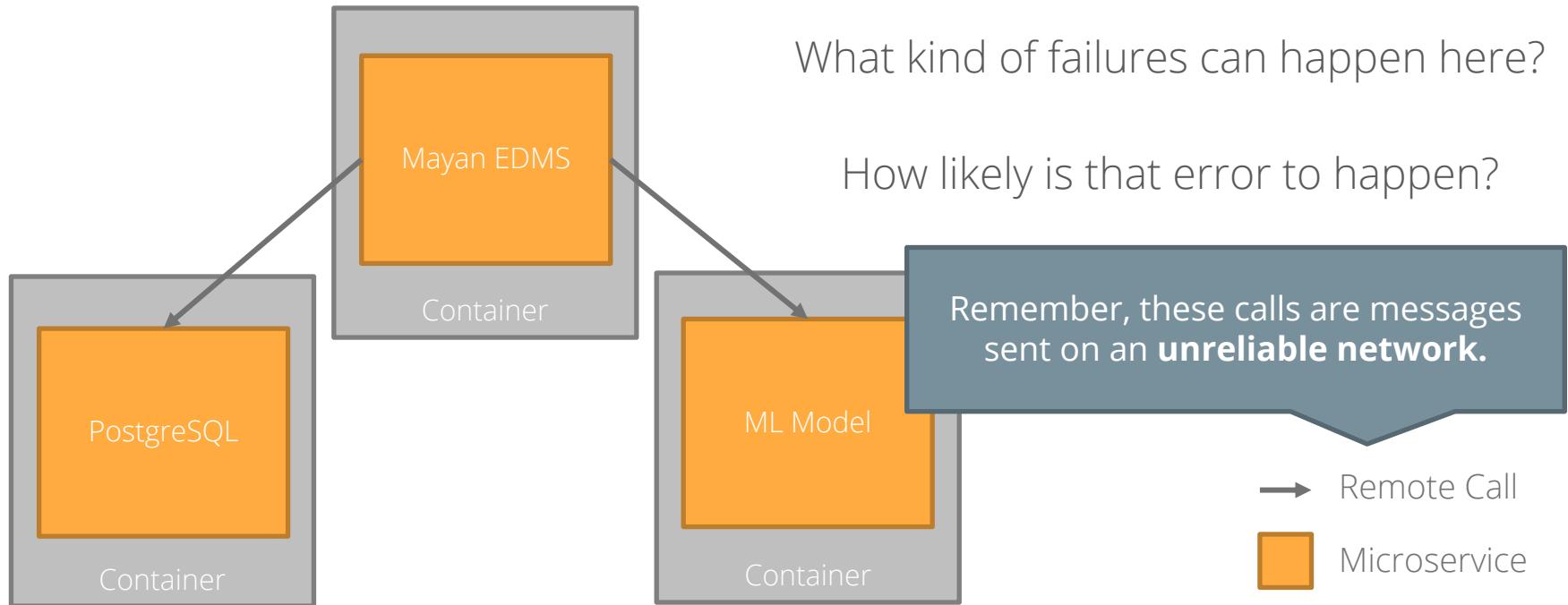
What kind of failures can happen here?

How likely is that error to happen?

How do I fix it?

→ Process Call  
■ Microservice

# Exercise: Microservice Application



# Failures in Microservice Architectures

1. Network may be partitioned
2. Server instance may be down
3. Communication between services may be
4. Server could be overloaded and response
5. Server could run out of memory or CPU

All of these issues  
**can be indistinguishable**  
from one another!

Making the calls across the network to  
multiple machines makes the  
probability that the system is operating  
under failure **much higher.**

These are the problems of  
**latency** and **partial failure**.

# Where Do We Start?

How do we even begin to test these scenarios?

Is there any **software** that can be used to test these types of failures?

Let's look at a **few ways** companies do this.



It's very much the Adulox shipping metric and something we're trying to do is auto context handling. Instead of trying to see the one or the largest vendor ship operations in the country, have systems take failed, and see the context of what happened. If it's a car seat and it's broken, then we can say, "Well, it's a car seat, so we can't just ignore it."

The reason is that if it's not a car seat, then it's probably not a car seat. But it's not a car seat, though it could have been. Instead, if it's an article designed

for a child, then it's probably a car seat. So if you're trying to figure out what happened, then you can't just ignore it. That's kind of the approach we're taking.

That's kind of the approach we're taking. It's kind of the approach of learning to embrace failure, but rather than learning to embrace failure after it's already happened, it's learning to embrace failure before it's even happened.

In March 2009, Amazon created *GoatCycle*, a program designed to increase resilience by periodically

injecting minor failures into critical systems, with the goal of detecting flaws and subtle dependencies that could lead to system-wide failures. The idea is that by injecting small amounts of failure into the system, more companies won't be taken off guard and start to fail when the next crisis comes. This document covers

more about how GoatCycle works and how it's being used at Amazon and other companies.

Partners include Drew Robbins, the architect of GoatCycle at Amazon, who has officially

left Amazon to work at Google. He is currently working on a project called "Project Resilience," which is based on the principles of incident response. It is focused on how to build a culture of resilience across an organization, and how to build a culture of resilience across an organization.

Other partners include Gregor Kilian, who makes GoatCycle available to other organizations, and Tim Linternell, who is the author of the book "Resilient Systems Engineering."

Gregor Kilian is the lead developer of GoatCycle. He has been with the program almost from the time it started in 2009. He also works on other resilience projects, most of which are focused on the cloud.

Joining this discussion is Gregor Kilian, Jason Robbins, John Allegretto, and Tim Linternell.

**TOM LINTERNELL:** Jason, you've probably been involved in more GoatCycle exercises than anybody.

That's the most important lesson you've taken away from this?

# Game Days

Our applications are built on and with “unreliable” components

Failure is inevitable (fraction of percent; at Google scale, ~multiple times)

Goals:

- Preemptively trigger the failure, observe, and fix the error
- Script testing of previous failures and ensure system remains resilient
- Build the necessary relationships between teams before disaster strikes



It's very much the Adobe shipping model - and something is wrong if a beta isn't handling interactions the way the final version will. Some systems have fails, and we have to make sure they're safe to land. If you can't do that, then you have to fix them before shipping the software.

The question is, is that it's not a risk if something could land here first? It's an exercise designed to teach a company how to adapt to the inevitable system failure. Thing is, most companies don't know what to do with it, so they're preparing for it & trying to accept it.

Most companies are afraid of failing, and that's a lack of having experience in failure. But failure is how we become better able to adapt to it when it occurs. Because experiencing a familiar concept at high stakes is a great way to learn from failure, and it's a great way to learn from success as well.

In March 2009, Amazon created *CrashBox*, a program designed to increase resilience by purposefully injecting minor failures into critical systems, some related to disk drives and cable disruptions.

For example, one of the first tests involved sending a signal to a server to indicate that a hard drive was failing. Most companies would just ignore the message, but more resilient ones see it earlier and start to alert their own systems. This discovery allows them to react faster.

Participants included Drew Robbins, the architect of CrashBox at Amazon, who he was officially promoted to Vice President of Reliability Engineering in 2010. He also founded the Vehicle Web Performance and Reliability group at Amazon, which is responsible for reliability engineering across the vehicle division. The division includes Amazon's vehicle delivery fleet, Amazon Fresh, and Amazon Prime Now.

This division's experience includes a failure of Amazon Fresh's delivery van during Prime Now's first delivery, which caused a significant delay in delivery times.

Participants also included Gregor Hohpe, the architect of CrashBox at Google, who has been with the program almost from the time it started in 2008. He also works on other infrastructure projects, most of which are focused on the search engine.

Joining this discussion is Gregor Hohpe, Jason Allspaw, who is an reliability engineer at Etsys Resilience Engineering team, and Tim Lomax, a principal engineer at LinkedIn, and author of *Time Management for Software Professionals* and *The Practice of System and Network Administration*.

**TOM OMBRECKE:** Jason, you've probably been involved in more CrashBox exercises than anybody. What's the most important lesson you've taken away from that?

Yes, and the exercise should be designed to make people feel a little uncomfortable. The truth is that **things often break in ways that people cannot possibly imagine**.



John Allspaw  
Former CTO  
Etsy

I've got a crazy story...



Kelly Sommers  
Cassandra MVP

# Google GameDays

Experiments take roughly 24 – 96 hours:

- 00 - 24h: the initial response, the appearance of the 'big' problems.
  - 24 - 48h: team to team testing and response, bi-directional testing
  - 72 - 96h: exhaustion; part of the test to identify the human response

...in a real emergency, you might not have the option of handing off work at the end of your shift.



Kripa Krishnan  
Director, Cloud Ops & Site Reliability Engineering  
Google

# AWS Outage: April 21, 2011

EC2 outage in us-east-1 (Northern Virginia)

Outage affects:

- Foursquare
- Quora
- Reddit

Outage results in performance problems and in some cases data loss

# AWS Outage: April 21, 2011

At 12:47 AM PDT on April 21st, a **network change** was performed as part of our normal AWS scaling activities in a single Availability Zone in the US East Region. The configuration change was to upgrade the capacity of the primary network. During the change, one of the standard steps is to shift traffic off of one of the redundant routers in the primary EBS network to allow the upgrade to happen. The traffic shift was executed incorrectly and rather than routing the traffic to the other router on the primary network, the traffic was routed onto the lower capacity redundant EBS network. For a portion of the EBS cluster in the affected Availability Zone, this meant that they did not have a functioning primary or secondary network because traffic was purposely shifted away from the primary network and **the secondary network couldn't handle the traffic level it was receiving**. As a result, many EBS nodes in the affected Availability Zone were completely isolated from other EBS nodes in its cluster. Unlike a normal network interruption, this change disconnected both the primary and secondary network simultaneously, leaving the affected nodes completely isolated from one another.

# AWS Outage: April 21, 2011

When this network connectivity issue occurred, a large number of EBS nodes in a single EBS cluster lost connection to their replicas. When the incorrect traffic shift was rolled back and network connectivity was restored, these nodes rapidly began searching the EBS cluster for available server space where they could re-mirror data. Once again, in a normally functioning cluster, this occurs in milliseconds. In this case, because the issue affected such a large number of volumes concurrently, the free capacity of the EBS cluster was quickly exhausted, leaving many of the nodes “stuck” in a loop, continuously searching the cluster for free space. This quickly led to a “re-mirroring storm,” where a large number of volumes were effectively “stuck” while the nodes searched the cluster for the storage space it needed for its new replica. At this point, about 13% of the volumes in the affected Availability Zone were in this “stuck” state.

## Primary Outage

At 12:47 AM PDT on April 21st, a network change was performed as part of our normal AWS scaling activities in a single Availability Zone in the US East Region. The configuration change was to upgrade the capacity of the primary network. During the change, one of the standard steps is to shift traffic off of one of the redundant routers in the primary EBS network to allow the upgrade to happen. The traffic shift was executed incorrectly and rather than routing the traffic to the other router on the primary network, the traffic was routed onto the lower capacity redundant EBS network. For a portion because traffic was purposely shifted away from the affected Availability Zone were completely isolated from the network simultaneously, leaving the affected nodes disconnected.

When this network connectivity issue occurred, a large network connectivity was restored, these nodes rapidly cluster, this occurs in milliseconds. In this case, because many of the nodes "stuck" in a loop, continuously search while the nodes searched the cluster for the storage system.

After the initial sequence of events described above, I entered the re-mirroring storm and exhausted its available volume API in particular) was configured with a long timeline has a regional pool of available threads it can use had no ability to service API requests and began to fail disabled all new Create Volume requests in the affected

Two factors caused the situation in this EBS cluster to fail when they could not find space, but instead, continue to fail when they were concurrently closing a large number of volumes. However, during this re-mirroring storm, the volume controller bug, resulting in more volumes left needing to be mirrored.

By 5:30 AM PDT, error rates and latencies again increased for the EC2 instance, the EBS nodes with the volume data, and the replica recognized by the EC2 instance as the planned race condition described above, the volume of calls increased as the system retried and new requests were made. The team began disabling all communication between the affected Availability Zone (we will discuss recovery of the system later).

A large majority of the volumes in the degraded EBS team developed a way to prevent EBS servers in the other essential communication between nodes in the becoming "stuck". Before this change was deployed, the becoming "stuck". However, volumes were also slowly that when this change was deployed, the total "stuck"

Customers also experienced elevated error rates until zone. This occurred for approximately 11 hours, from were able to create EBS-backed EC2 instances but were in the EBS control plane that is only needed for attaching errors were overshadowed by the general error from the EBS-backed EC2 instances declined rapidly and returned

## Recovering EBS in the Affected Availability Zone

By 12:04 PM PDT on April 21st, the outage was contained to the one Availability Zone and the degraded EBS cluster was stabilized. APIs were working well for all other Availability Zones and additional volumes were no longer becoming "stuck". Our focus shifted to completing the recovery. Approximately 13% of the volumes in the Availability Zone remained "stuck" and the EBS APIs were disabled in that one affected Availability Zone. The key priority became bringing additional storage capacity online to allow the "stuck" volumes to find enough space to create new replicas.



or reuse the failed node until every data replica is successfully replicated. We did not want to re-purpose this capacity until we were sure we had the capacity to replace that capacity in the cluster. This required the time-to-capacity into the degraded EBS cluster. Second, because of the size of the cluster in the step described above), the team had difficulty to allow negotiation to occur with the newly-built servers without causing the team to have to navigate a number of issues as they worked through significant amounts of new capacity and working through the replication process. Availability Zone 1 was restored by 12:30PM PDT on April 10, 2011. The remaining Availability Zones were restored by 12:30PM PDT on April 11, 2011. The majority of the attached EC2 instances became accessible again at this time.

cess to the affected Availability Zone and restoring access to the failed EBS nodes to the EBS control plane and vice versa. This effort API access online to the impacted Availability Zone centered on instance of the EBS control plane, one we could keep partitioned. We rapidly developed throttles that turned out to be too coarse; the morning of April 23rd, we worked on developing finer-grain initial tests. Initial tests against the EBS control plane demonstrated finished enabling access to the EBS control plane to the degraded negotiate which replica would be writable, to once again be usable Availability Zone.

nes in the Region. The recovery of the remaining 2.2% of affected event as an extra precaution against data loss while the event began processing batches through the night. At 12:30 PM PDT fected volumes. At this point, the team began forensics on the 10 PM PDT, the team began restoring these. Ultimately, 0.07% of the

"RDS"). RDS depends upon EBS for database and log storage, and as

nultiple Availability Zones ("multi-AZ"). Single-AZ database instances did if one of the EBS volumes it was relying on got "stuck." In the relatively-bigger portion of the RDS population than the ases aggregate I/O capacity for database workloads under normal ll the volume is restored. The percentage of "stuck" single-AZ ed. The percentage of "stuck" single-AZ database instances in the hours, and the rest recovered throughout the weekend. Though we ity Zone had EBS storage volume that was not option to initiate point-in-time database restore operations.

RDS multi-AZ deployments provide redundancy by synchronously replicating data between two database replicas in different Availability Zones. In the event of a failure on the primary replica, RDS is designed to automatically detect the disruption and fail over to the secondary replica. Of multi-AZ database instances in the US East Region, 2.5% did not automatically failover after experiencing "stuck" I/O. The primary cause was that the rapid succession of network interruption (which partitioned the primary from the secondary) and "stuck" I/O on the primary replica triggered a previously un-encountered bug. This bug left the primary replica in an isolated state where it was not safe for our monitoring agent to automatically fail over to the secondary replica without risking data loss, and manual intervention was required. We are actively working on a fix to resolve this issue.

In very much the Adelphi shipping service, something is wrong if a data center handling interactions for one of the largest online retail operations in the country, home systems have failed, and no one has noticed. That's what makes resilience so important. It's not just about being able to handle things that are normal.

The question is, well, is it not a bit of a stretch though? I could have been thinking, if it's an exercise designed to teach a company how to adapt to the inevitable system failure, "Okay, break, disaster happens, failure is there, we've got to fix it." But I think that's not what resilience means. Resilience means that you can't prevent failure, but you're preparing for it & it's to accept it.

It's not about accepting failure, it's about accepting the reality of something important to failure, but rather to become better able to adapt to it when it occurs. Because experiencing a familiar concept in high-risk environments is a great way to learn how to deal with it, and that's what resilience is all about, I think.

In late 2009, Amazon created Gamelab, a program designed to increase resilience by purposely crashing major failures into critical systems, some related to discrete flaws and subtle dependencies.

Resilience is not just about being able to handle the unexpected, it's also about being able to handle the expected. Most companies now see the value and have started to adopt the same practices. This discussion covers some of those.

Participants include Jason Rabines, the architect of Gamelab at Amazon, who has officially joined the company; Gregor Kiczales, the author of the original paper on resilience engineering and principles of resilient systems; Tim Linternell from the Vehicle Web Performance and Reliability group at Ford Motor Company; and John Allegretto, the lead developer of the financial application, which makes up half a petabyte framework for infrastructure automation. Running Gamelab is a way to test the limits of resilience engineering. It's a way to test the limits of what resilience engineering can do. It's a way to test the limits of what resilience engineering can't do. It's a way to test the limits of resilience engineering from failure.

Joining the discussion is Tom Kyte, Kitzman, who has been with the program almost from the time it started in 2006. He also works on other resilience projects, most of which are focused on the cloud.

Monitoring this discussion is a Google+ community. Join Gamelab, which is an visibility request.

For more information on resilience engineering, see the book Resilience Engineering: Toward Sustainable High-Performance Work Systems by John M. Hollnagel, Peter Hancock, and William T.swanson.

**TOM LINTERNELL:** Jason, you've probably heard mentioned in more Gamelab exercises than anything else that the most important lesson you're take away from this.

1

# Cornerstones of Resilience

**"[resilient is the] ability to sustain operations before, during, and after an unexpected disturbance"**



1. Anticipation: know what to expect
2. Monitoring: know what to look for
3. Response: know what to do
4. Learning: know what just happened  
(e.g, postmortems)

# Anticipation

“[...] get people throughout the organization to start building their anticipation muscles by **thinking about what might possibly go wrong.**”



These experiments form a cycle where developers begin to anticipate what might possibly go wrong during development, which adds to the overall resilience of the system.

It's very much the Airbus shipping metric and something we're trying at a lot of contact handling interactions for the one of the largest airline operators in the country. Some systems have failed, and we've had to do some work to make sure that they're more robust, and that's been a good exercise in learning how to do things differently.

The first question is: But it's not a real disruption though? I could have fixed it? It's an exercise designed to teach a company how to adapt to the inevitable system failure. Thing is, most failures follow is that, and if you can't fix it, then you're probably not prepared for it to happen. If you can't fix it, then you can't prevent failure, but by preparing for it, it's easier to accept it.

It's also an exercise in learning to embrace failure. Because it's a way of learning important to failure, but rather than become better able to adapt to when it occurs, because experiencing a familiar concept at high stakes is a great way to learn, and it's a great way to learn how to embrace failure and how to learn from failure as well.

In March 2009, American created *GoToZero*, a program designed to increase resilience by periodically shutting major failures into critical systems, some regularly to discover flaws and enable identification of potential problems before they become serious. The idea is that by exposing teams to failure, they will learn how to handle it better and how to react to other types of events. This discipline creates more resilience across the organization.

Participants include Steve Rabkin, the architect of GoToZero, who has effectively created a culture of failure at American. Steve has been involved in the development of safety principles of incident response. He left American in 2006 and joined the Vehicle Test Performance and Safety department at Ford Motor Company, where he developed the Ford Safety System, a system of functional tests, which makes it a great platform for infrastructure automation. Boeing Computer Services, the IT arm of Boeing, has adopted a similar approach to resilience training, called Resilient Systems Engineering. This approach experience includes a failure and prevention tool, joining Ford as engineering resilience training.

Steve Rabkin is currently working at the University of California Berkeley, where he is involved in the development of resilience training for the aerospace industry.

Gregor Hohpe is a member of Google's team in Berlin, Germany, who has been with the program almost from the time it started in 2009. He also works on other infrastructure projects, most of which are focused on the cloud and distributed systems.

Joining this discussion is Gregor Hohpe, from Germany, who is a reliability engineer that has worked on the design of distributed systems, and has been involved in the development of the Resilient Systems Engineering for Boeing, Infrastructure and IT Practices of System and Network Administration.

**TOM LINTERNELL** Jason, you've probably been involved in more GoToZero exercises than anybody. What's the most important lesson you've taken away from that?





## Chaos Engineering

By Brian Wawak, Reed Johnson, Luke Kanies, Scott MacVittie, and Casey Overmyer

This article originated in an internal discussion at Netflix about how to increase reliability. For months, we worked on ways to make our system more reliable. One day, we decided to try something different: chaos engineering. We chose Netflix because it has a culture of experimentation and a desire to learn from failure.

**Modern software-based services are implemented in complex distributed systems. These systems are often deployed in environments that are highly dynamic and unreliable. Chaos engineering uses experimentation and automation to identify and mitigate potential problems before they become critical. This article describes the principles of chaos engineering that we have learned at Netflix and our experiences.**



**HARRY YEOMAN, JON CLEW, AND SCOTT MACVITIIE** are on a mission to make availability, reliability, and performance better for everyone at Netflix. They are working on the Netflix Chaos Monkey project, which was created to test the resilience of the company's distributed systems. The team is also involved in the Netflix DevOps culture, which emphasizes continuous improvement and experimentation. They believe that by understanding the challenges of building and maintaining complex systems, they can help others do the same. They are also involved in the Netflix Open Source Project, which aims to share their knowledge and expertise with the wider developer community.

PHOTOGRAPH BY DAVID RYAN

# Netflix: Background

Started as a DVD-by-mail business because Reed Hastings was annoyed with Blockbuster late fees

**Problem:** when new movies come out, there's only hundreds of DVDs to service multiple thousands of demand

Stream movies instead of purchasing and mailing DVDs out to customers

**Problem:** must purchase enough compute to handle peaks (7pm+ weekends) vs valleys (noon, weekday)

## Chaos Engineering

By Matt Zaharia, Brian de Roos, Luke Kanies, Scott MacVittie, and Casey Rutherford

Modern software-based services are implemented using microservices, which are deployed in a continuous delivery mode. Chaos engineering uses experimentation to validate the reliability of these systems. This article describes the principles of chaos engineering that drove our how to design and run experiments.



MATT ZAHARIA, Inc. One

For us, resilience has always been a key priority. We have invested in a culture of reliability. For example, we have a culture of self-healing where we can automatically fix errors. This is something that we have done for years now.

Modern software-based services are implemented using microservices, which are deployed in a continuous delivery mode. Chaos engineering uses experimentation to validate the reliability of these systems. This article describes the principles of chaos engineering that drove our how to design and run experiments.

[Read the full article](#)

PHOTO: ANDREW HETHERINGTON

NETFLIX

# Netflix: Cloud Computing

Significant deployment in Amazon Web Services in order to remain elastic in times of high and low load (first public, 100% w/o content delivery.)

Pushes code into production and modifies runtime configuration hundreds of times a day

a customer who can't watch a video because of a service outage might not be a customer for long.

Key metric: availability



“Chaos Engineering”  
Basiri *et al.*, IEEE Software 2016

## Chaos Engineering

By Brian Wernick, Scott Beaumont, and Casey Broderick, Netflix

This series, organized in an effort to highlight the most interesting and useful approaches to improving system reliability. For example, we look at how Netflix uses Chaos Monkey to test its cloud infrastructure, and how D2C shows, Company Cloud, and other companies have adopted similar approaches.



**Modern software-based services are implemented across multiple regions and data centers. In such a distributed system, it's important to understand how failures occur. Chaos engineering uses experimentation to identify potential failure points and validate the resilience of a system. This article describes the principles of chaos engineering that drove our team to design and run experiments.**

**HARRY YEAM AND JON CLEAVER** are software engineers at Netflix. They are both members of the Netflix Simian Army, which is responsible for the development and maintenance of the Chaos Monkey, Chaos Kong, and Latency Monkey tools. They are also members of the Netflix DevOps team, which is responsible for the automation of deployment and monitoring of Netflix's production systems. They have been working at Netflix for over five years.

**Planning & Perspectives** is a regular column in *ACM SIGARTICLES* that highlights the latest research and development in the field of artificial intelligence and robotics.

**IN THIS ISSUE** includes a column by Harry Yeam and Jon Cleaver on Chaos Engineering.

# Chaos Engineering: The History

Experimentation to build confidence around a system to withstand turbulent conditions in produ

Chaos Monkey has proven successful; today all Netflix engineers design their services to handle instance failures as a matter of course.

Netflix's Simian Army

- (*the original*) Chaos Monkey:  
Randomly **terminates** EC2 instances in production
- Chaos Kong:  
**Simulates** the failure of an entire EC2 region in AWS
- Latency Monkey:  
**Injects latency** to simulate  
react appropriately

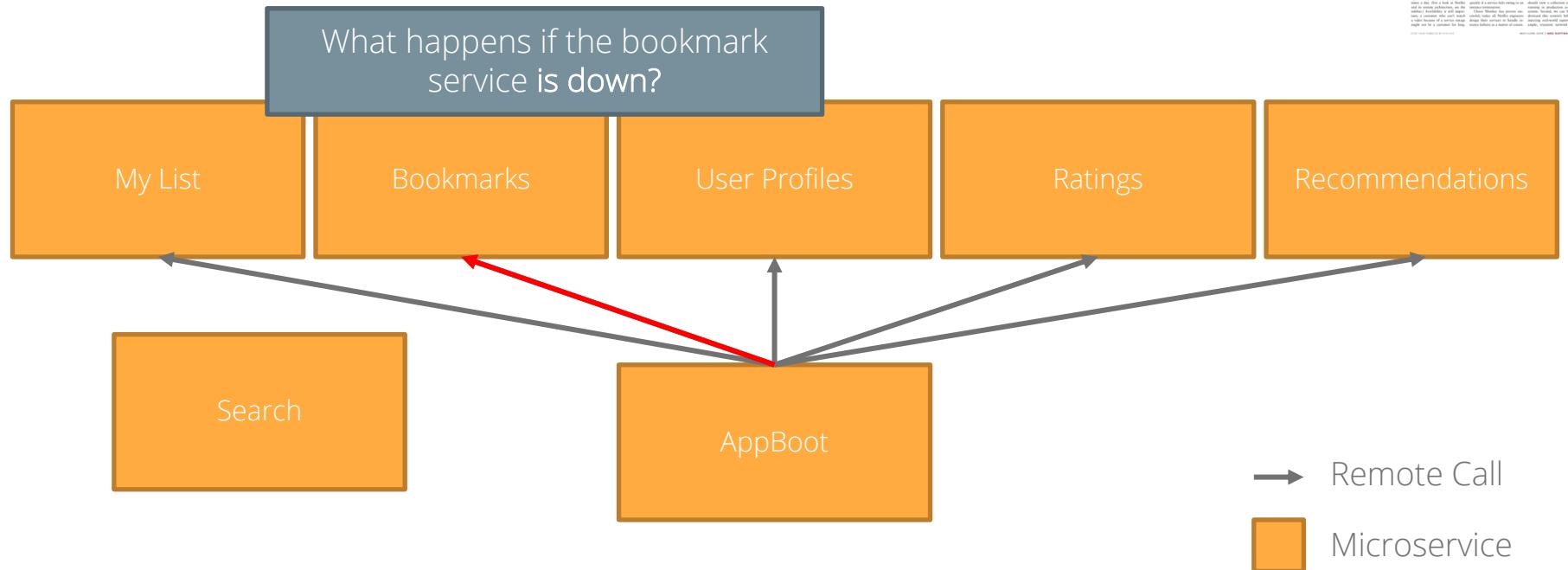


Have Chaos Monkey crash development instances, too!



# Netflix UI: AppBoot

What happens if the bookmark service is down?



FOCUS: DEVOPS

## Chaos Engineering

By Brian Hayes, Daniel Rizzo, Luke Kanies, Scott MacVittie, and Cadey Pfeiffer, Netflix

Modern software-based services are implemented in complex, distributed systems. In such systems, failure modes, Chaos engineering uses experimentation to identify potential failure points and develop principles of chaos engineering that describe how to design and run experiments.

MANY YEARS AGO, Jim Gray, the author of many books on databases, asked me what I thought was the most important problem in distributed systems. I told him that it was the lack of a clear definition of what a distributed system is. He responded that he had no problem with that. "A distributed system is one in which the failure of a computer you didn't even know existed can affect the operation of your program," he said. "That's a distributed system." I responded that I agreed with him, but that I also believed that a distributed system is one in which the performance of a service or application depends on the behavior of other components that are not under your control. I believe that there are two types of distributed systems. One type is a distributed system that is designed to work with other components in a distributed environment. The other type is a distributed system that is designed to work with other components in a distributed environment. The difference between the two types is that the first type is designed to work with other components in a distributed environment. The second type is designed to work with other components in a distributed environment.

PHOTO: ANDREW HETHERINGTON

# Principles of Chaos Engineering

1. Build a hypothesis around steady state behavior
2. Vary real-world events  
experimental events, crashes, etc.
3. Run experiments in production  
control group vs. experimental group  
~~draw conclusions to validate hypothesis~~  
However, “works properly” is too vague a basis for designing experiments.
4. Automate experiments to run continuously

Does everything seem to be working properly?

Are users complaining?



# Graceful Degradation: Anticipating Failure

Allow the system to degrade in a way it's still usable

## Fallbacks:

- Cache miss due to failure of cache;
  - Go to the bookmarks service and use value at possible latency penalty

Personalized content, use a reasonable default instead:

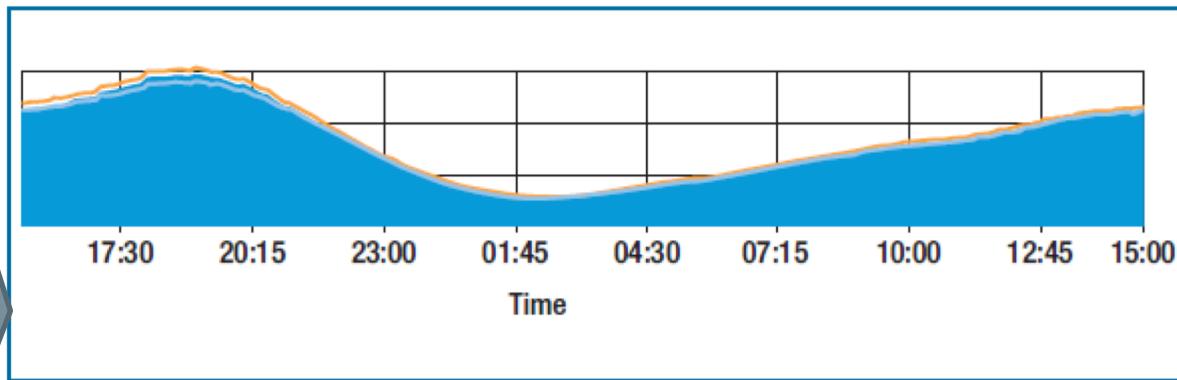
- What happens if recommendations are unavailable?
  - What happens if bookmarks are unavailable?



# Steady State Behavior

## Back to quality attributes: availability!

SPS is the primary indicator of the system's overall health.



**FIGURE 2.** A graph of SPS ([stream] starts per second) over a 24-hour period. This metric varies slowly and predictably throughout a day. The orange line shows the trend for the [redacted]. Ultimately, what we care about is whether users can find

mately, what we care about is whether users can find content to watch and successfully watch it.



# Exercise: Quality Attributes

1. What would a quality attribute be for an **e-commerce website** to characterize the steady-state behavior of the system?
  2. What would a quality attribute be for an **advertisement platform** to characterize the steady-state behavior of the system?
  3. What would a quality attribute be for an **admissions system** to characterize the steady-state behavior of the system?

## Chaos Engineering

By Michael Bevilacqua, Paul de Ruiter, Luke Kanies, Scott MacVittie, and Casey Rutherford, Netflix

[Read the full article](#)

This series emerged as an effort to share the approach we've taken at Netflix to increase system reliability. For months, we've been working with our engineering teams to build a culture around SRE. Now, Company Cloud has joined forces with Netflix's DevOps team to bring you these stories in which we'll share how Netflix has approached the challenges of building reliable systems.

**Modern software-based services are implemented in complex ways. In fact, they're often deployed in broken modes. Chaos engineering uses experimentation to identify and mitigate these risks. This article describes a practical approach to chaos engineering that describes how to design and run experiments.**



MICHAEL BEVILACQUA, PAUL DE RUITER, LUKE KANIES, SCOTT MACVITIE, AND CASEY RUTHERFORD, NETFLIX

[Read the full article](#)

**Editor's Note:** Netflix's approach to chaos engineering is one of the most well-known and widely adopted in the industry. This article provides a detailed overview of their methodology and best practices.

**Planning & Perspectives**

Is the time right for your organization to embrace chaos engineering? This article provides a practical guide to help you determine if it's the right fit for your organization.

**Read the full article**

# Making Hypotheses

## No trivial hypotheses

- Overloading the system will increase the CPU, etc.
- Hypothesis should be made w.r.t overall system health metric

## Monitor finer-grained metrics

- Monitor the CPU, other resources
- Indicators of degraded mode operation, etc.
- Use alerting to identify these issues to catch them early and anticipate



# Sampling of Netflix's Candidate Faults

1. Terminate virtual machine instances
  2. Inject latency into requests between different services
  3. Fail requests between services
  4. Fail an entire service
  5. Make an entire Amazon region unavailable

# Two Example Netflix Errors

1. Server is overloaded and takes longer and longer to respond  
Clients requests are placed in a queue to be serviced  
Local queue becomes exhausted, run out of memory  
Client service crash
  2. Client makes a request to a server that uses a cache  
Error (*transient*) is returned to the client  
Server caches the error  
Future clients read the cached error value

**Chaos Engineering**  
By Steve Lohr  
Labs Editor, Science News  
Labs Editor, Science News  
**J**Modern software-based services are implemented as complex systems of interconnected nodes. Failure modes, though often unpredictable, can have serious consequences. One way to develop resilience is through the well-known and developing principle of chaos engineering that directly tests the system's reliability.

# Testing Timeouts

Request to service waits 10 seconds before returning error to the user

Which messages do we delay?

- There are two options: the **request** or the **response**
- **Which** do we delay and **why**?

How **long** do we delay the message?

- There are **multiple correct answers** when choosing how long to delay



**FOCUS: DEVOPS**

## Chaos Engineering

By Michael Housley, Scott Beaumont, and Cindy Provenzano, Netflix

This series, organized by us at Netflix, explores how we've used chaos engineering to increase system reliability. For now, we're focusing on the basics of chaos engineering, such as how to identify potential failure points in your system. Next month, we'll look at how to measure the impact of those failures.

**Modern software-based services are implemented using microservices architectures, which makes them more complex and prone to failure. Chaos engineering can help you identify potential failure points in your system. Here's how to do it.**

**MICHAEL HOUSLEY** is a principal engineer at Netflix. **SCOTT BEAUMONT** is a senior principal engineer at Netflix. **CINDY PROVENZANO** is a principal engineer at Netflix.

**Editor's Note:** This article is part of a series on chaos engineering. See also "How Netflix Uses Chaos Engineering to Improve System Reliability," page 32.

**Photo: Netflix**

**CHASING PREDATORS**

In the last issue, we introduced the concept of chaos engineering, a discipline that uses controlled, safe, and repeatable system failures to identify potential problems before they become real ones. In this issue, we'll look at how to measure the impact of those failures.

**HARRY YEOMAN AND JON CLEARY** are authors of *Chaos Engineering: How Netflix Improves System Reliability by Injecting Failure* (O'Reilly Media). They are both Netflix engineers, as well as authors of the book *“How Netflix Improves System Reliability by Injecting Failure”*.

**CHASING PREDATORS**

In the last issue, we introduced the concept of chaos engineering, a discipline that uses controlled, safe, and repeatable system failures to identify potential problems before they become real ones. In this issue, we'll look at how to measure the impact of those failures.

**HARRY YEOMAN AND JON CLEARY** are authors of *Chaos Engineering: How Netflix Improves System Reliability by Injecting Failure* (O'Reilly Media). They are both Netflix engineers, as well as authors of the book *“How Netflix Improves System Reliability by Injecting Failure”*.

# How to run a Chaos Experiment

1. Define steady-state as some measurable output of a system that indicates normal behavior
2. Hypothesize that this steady state will continue in both the control group and experimental group
3. Introduce variables that reflect real-world events such server crashes, hard drives malfunctioning, and network connections being severed
4. Try to disprove the hypothesis by looking for a difference in steady state between the control group and the experimental group.

# Recitation

Run a **chaos experiment** using Mayan EDMS.

Come up with a hypothesis.

Run a chaos experiment for real using Gremlin!

# My Favorite Bug: Doomstones

1. Cassandra is a highly-available, eventually consistent database  
Accepts writes during network partitions
2. To determine what value to keep when nodes disagree after the network has healed, the client provides a clock with each write  
and the object with the newest recent clock wins.
3. To delete a value, you write a tombstone.
4. What happens when if delete a value and your computers clock is wrong? (it's incorrectly set in the future.)