

Chaos Engineering

Building confidence in your application and team
through failure experimentation

Christopher Meiklejohn

October 29, 2020

Administrivia

Homework 4B was due Nov 3rd, **now the 4th**.

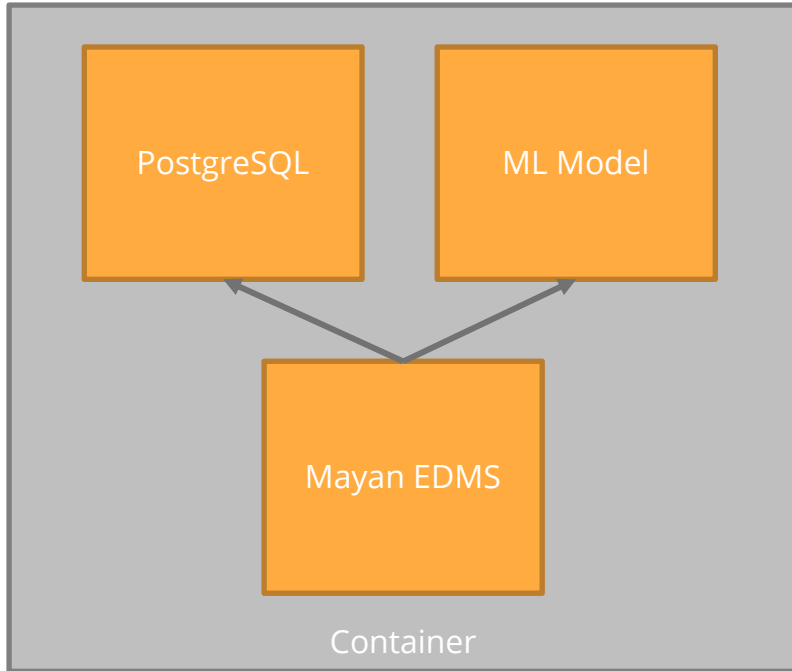
Homework 4C was due Nov 5th, **now the 6th**.

Learning Goals

Identify the need for chaos and resilience engineering

Understand the principles of chaos engineering

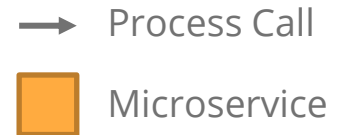
Exercise: Monolithic Application



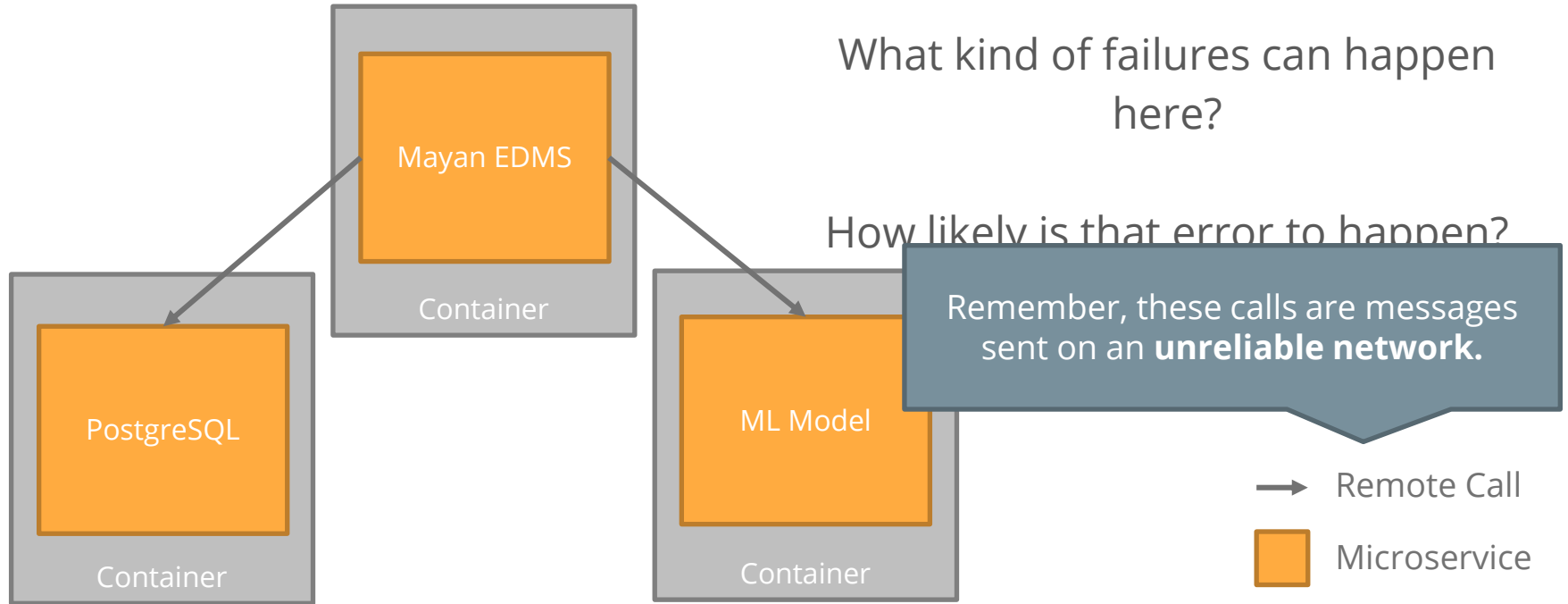
What kind of failures can happen here?

How likely is that error to happen?

How do I fix it?



Exercise: Microservice Application



Failures in Microservice Architectures

1. Network may **be partitioned**
2. Server instance **may be down**
3. Communication between services may **be delayed**
4. Server **could be overloaded** and response time **could be long**
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.

Game Days

Purposely **injecting failures** into critical systems in order to:

- Identify **flaws** and “latent defects”
- Identify **subtle dependencies** (which may or may not lead to a flaw/defect)
- Prepare a **response** for a disastrous event

Comes from “resilience engineering” typical in high-risk industries

Practiced by Amazon, Google, Microsoft, Etsy, Facebook, Flickr, etc.

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 common for the media to report on system outages, but it's less common to hear about the lessons learned from those outages. In one of the largest and most complex IT environments in the world, how do you learn from those outages and how do you prevent them from happening again?

It's not about preventing a failure, it's about preventing a failure from being a failure. It's not about preventing a failure from being a failure, it's about preventing a failure from being a failure.

It's not about preventing a failure, it's about preventing a failure from being a failure. It's not about preventing a failure from being a failure, it's about preventing a failure from being a failure.

It's not about preventing a failure, it's about preventing a failure from being a failure. It's not about preventing a failure from being a failure, it's about preventing a failure from being a failure.

It's not about preventing a failure, it's about preventing a failure from being a failure. It's not about preventing a failure from being a failure, it's about preventing a failure from being a failure.

It's not about preventing a failure, it's about preventing a failure from being a failure. It's not about preventing a failure from being a failure, it's about preventing a failure from being a failure.

It's not about preventing a failure, it's about preventing a failure from being a failure. It's not about preventing a failure from being a failure, it's about preventing a failure from being a failure.

It's not about preventing a failure, it's about preventing a failure from being a failure. It's not about preventing a failure from being a failure, it's about preventing a failure from being a failure.

It's not about preventing a failure, it's about preventing a failure from being a failure. It's not about preventing a failure from being a failure, it's about preventing a failure from being a failure.

It's not about preventing a failure, it's about preventing a failure from being a failure. It's not about preventing a failure from being a failure, it's about preventing a failure from being a failure.

It's not about preventing a failure, it's about preventing a failure from being a failure. It's not about preventing a failure from being a failure, it's about preventing a failure from being a failure.

It's not about preventing a failure, it's about preventing a failure from being a failure. It's not about preventing a failure from being a failure, it's about preventing a failure from being a failure.

Amazon.com Inc. (Amazon) is a leading e-commerce company that has grown to become one of the world's largest companies. In 2014, Amazon.com Inc. reported a revenue of \$136 billion, up from \$107 billion in 2013. Amazon.com Inc. is a public company listed on the NASDAQ stock exchange under the ticker symbol AMZN.

Amazon.com Inc. is a leading e-commerce company that has grown to become one of the world's largest companies. In 2014, Amazon.com Inc. reported a revenue of \$136 billion, up from \$107 billion in 2013. Amazon.com Inc. is a public company listed on the NASDAQ stock exchange under the ticker symbol AMZN.

Amazon.com Inc. is a leading e-commerce company that has grown to become one of the world's largest companies. In 2014, Amazon.com Inc. reported a revenue of \$136 billion, up from \$107 billion in 2013. Amazon.com Inc. is a public company listed on the NASDAQ stock exchange under the ticker symbol AMZN.

Amazon.com Inc. is a leading e-commerce company that has grown to become one of the world's largest companies. In 2014, Amazon.com Inc. reported a revenue of \$136 billion, up from \$107 billion in 2013. Amazon.com Inc. is a public company listed on the NASDAQ stock exchange under the ticker symbol AMZN.

Amazon.com Inc. is a leading e-commerce company that has grown to become one of the world's largest companies. In 2014, Amazon.com Inc. reported a revenue of \$136 billion, up from \$107 billion in 2013. Amazon.com Inc. is a public company listed on the NASDAQ stock exchange under the ticker symbol AMZN.

Amazon.com Inc. is a leading e-commerce company that has grown to become one of the world's largest companies. In 2014, Amazon.com Inc. reported a revenue of \$136 billion, up from \$107 billion in 2013. Amazon.com Inc. is a public company listed on the NASDAQ stock exchange under the ticker symbol AMZN.

Amazon.com Inc. is a leading e-commerce company that has grown to become one of the world's largest companies. In 2014, Amazon.com Inc. reported a revenue of \$136 billion, up from \$107 billion in 2013. Amazon.com Inc. is a public company listed on the NASDAQ stock exchange under the ticker symbol AMZN.

Amazon.com Inc. is a leading e-commerce company that has grown to become one of the world's largest companies. In 2014, Amazon.com Inc. reported a revenue of \$136 billion, up from \$107 billion in 2013. Amazon.com Inc. is a public company listed on the NASDAQ stock exchange under the ticker symbol AMZN.

Amazon.com Inc. is a leading e-commerce company that has grown to become one of the world's largest companies. In 2014, Amazon.com Inc. reported a revenue of \$136 billion, up from \$107 billion in 2013. Amazon.com Inc. is a public company listed on the NASDAQ stock exchange under the ticker symbol AMZN.

Amazon.com Inc. is a leading e-commerce company that has grown to become one of the world's largest companies. In 2014, Amazon.com Inc. reported a revenue of \$136 billion, up from \$107 billion in 2013. Amazon.com Inc. is a public company listed on the NASDAQ stock exchange under the ticker symbol AMZN.

Amazon.com Inc. is a leading e-commerce company that has grown to become one of the world's largest companies. In 2014, Amazon.com Inc. reported a revenue of \$136 billion, up from \$107 billion in 2013. Amazon.com Inc. is a public company listed on the NASDAQ stock exchange under the ticker symbol AMZN.

Amazon.com Inc. is a leading e-commerce company that has grown to become one of the world's largest companies. In 2014, Amazon.com Inc. reported a revenue of \$136 billion, up from \$107 billion in 2013. Amazon.com Inc. is a public company listed on the NASDAQ stock exchange under the ticker symbol AMZN.

Example: Amazon GameDay

Full data center destruction (Amazon EC2 region)

- Not all failures can be actually performed and must be **simulated!**
- Only a few days notice (months) that a GameDay **will be happening**
- **Real failures in the production environment**

Discovered **latent defect** where the monitoring infrastructure responsible for detecting errors and paging employees **was located in the zone of the failure!**

It was one of the earlier, slightly smaller-scale meetings we attended at an annual team building event. It was the first time we had a chance to sit down with the folks who had built and run the system for us. They had a lot of time and space to prepare and speak to us about the system they were building.

It was a great first experience, but it was also a little bit of a shock. I had never designed a system before, and I was sitting in a room with people who had. They had a lot of experience, and I was sitting in a room with people who had a lot of experience. It was a great first experience, but it was also a little bit of a shock. I had never designed a system before, and I was sitting in a room with people who had.

It was a great first experience, but it was also a little bit of a shock. I had never designed a system before, and I was sitting in a room with people who had. They had a lot of experience, and I was sitting in a room with people who had a lot of experience. It was a great first experience, but it was also a little bit of a shock. I had never designed a system before, and I was sitting in a room with people who had.

It was a great first experience, but it was also a little bit of a shock. I had never designed a system before, and I was sitting in a room with people who had. They had a lot of experience, and I was sitting in a room with people who had a lot of experience. It was a great first experience, but it was also a little bit of a shock. I had never designed a system before, and I was sitting in a room with people who had.

It was a great first experience, but it was also a little bit of a shock. I had never designed a system before, and I was sitting in a room with people who had. They had a lot of experience, and I was sitting in a room with people who had a lot of experience. It was a great first experience, but it was also a little bit of a shock. I had never designed a system before, and I was sitting in a room with people who had.

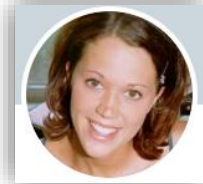
It was a great first experience, but it was also a little bit of a shock. I had never designed a system before, and I was sitting in a room with people who had. They had a lot of experience, and I was sitting in a room with people who had a lot of experience. It was a great first experience, but it was also a little bit of a shock. I had never designed a system before, and I was sitting in a room with people who had.

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

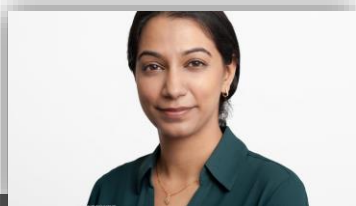
A discussion with Anna Bahram, Kripa Krishnan, Sam Allgeier, and Tom Linneman
It's common for the industry to glorify system availability and reliability, but Google's approach to handling outages is a stark contrast. In one of the largest articles ever published in the industry, Google's Site Reliability Engineering team shares their philosophy on how to handle outages. The article is a must-read for anyone interested in system reliability and incident response. It's a great read for anyone who wants to learn how to handle outages in a production environment. The article is a must-read for anyone who wants to learn how to handle outages in a production environment. The article is a must-read for anyone who wants to learn how to handle outages in a production environment.

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 of the outage, because traffic was purposely shifted away from the primary network, the affected Availability Zone were completely isolated from the primary network simultaneously, leaving the affected nodes cut off from the primary network.

When this network connectivity issue occurred, a large number of nodes in the affected Availability Zone were completely isolated from the primary network. When network connectivity was restored, these nodes rapidly returned to the primary network. In this case, because of the network connectivity issue, many of the nodes "stuck" in a loop, continuously searching for the storage service while the nodes searched the cluster for the storage service.

After the initial sequence of events described above, the affected Availability Zone entered the re-mirroring storm and exhausted its available capacity. Its available volume API (in particular) was configured with a long timeout. Because the volume plane has a regional pool of available threads it can use, it had no ability to service API requests and began to fail. This led to disabled all new Create Volume requests in the affected Availability Zone.

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

By 5:30 AM PDT, error rates and latencies again increased. At this point, EC2 instance, the EBS nodes with the volume data, are not recognized by the EC2 instance as the place where the race condition described above, the volume of calls increased as the system retried and new request queue began disabling all communication between the affected Availability Zone (we will discuss recovery of the affected Availability Zone).

A large majority of the volumes in the degraded EBS cluster were not essential communication between nodes in the affected Availability Zone becoming "stuck". Before this change was deployed, the affected Availability Zone becoming "stuck". However, volumes were also slowly becoming "stuck" that when this change was deployed, the total "stuck" volumes increased.

Customers also experienced elevated error rates until the primary network was restored. This occurred for approximately 11 hours, from 12:47 AM to 1:58 AM PDT. We were experiencing significantly-elevated error rates and latencies. New EBS-backed EC2 launches were being affected by a specific API in the EBS control plane that is only needed for attaching new instances to volumes. Initially, our alarming was not fine-grained enough for this EBS control plane API and the launch errors were overshadowed by the general error from the degraded EBS cluster. At 11:30 AM PDT, a change to the EBS control plane fixed this issue and latencies and error rates for new EBS-backed EC2 instances declined rapidly and returned to near-normal at Noon PDT.



Recovering EBS in the Affected Availability Zone

By 12:04 PM PDT on April 21st, the outage was contained to the one affected 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.



not reuse the failed node until every data replica is successfully re-created. We did not want to re-purpose this failed capacity until we were sure we had enough capacity to replace that capacity in the cluster. This required the time to create new capacity into the degraded EBS cluster. Second, because of the outage, the team had difficulty to allow negotiation to occur with the newly-built servers without the team had to navigate a number of issues as they worked to create significant amounts of new capacity and working through the replication lag. Affected Availability Zone were restored by 12:30PM PDT on April 21st. The majority of the attached EC2 instances because some were blocked from writing and elect a new writable copy.

Access to the affected Availability Zone and restoring access to the degraded EBS nodes to the EBS control plane and vice versa. This effort required the team to get API access online to the impacted Availability Zone centered on the degraded EBS control plane, one we could keep partitioned to the degraded EBS control plane, one we could keep partitioned to the degraded EBS control plane that turned out to be too coarse-grained. The morning of April 23rd, we worked on developing finer-grained access. Initial tests of traffic against the EBS control plane demonstrated that the team finished enabling access to the EBS control plane to the degraded Availability Zone which replica would be writable, to once again be usable by the affected Availability Zone.

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

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

multiple Availability Zones ("multi-AZ"). Single-AZ database instances are not as resilient as multi-AZ instances. 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.

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.

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)

It's common for the media to report on the success of a company or organization. However, it's less common to hear about the challenges they face. In this case study, we explore the challenges of resilience and how to build it into your organization.

The goal was to build a resilient organization that could handle any event. It was a complex task, but it was worth the effort. The organization was able to sustain operations during a major disaster, and it was able to recover quickly.

There are many ways to build resilience. One way is to invest in training and development. Another way is to invest in technology. A third way is to invest in people. All of these things are important, and they all need to be done.

Resilience is a complex concept, and it's not always easy to understand. However, it's a concept that is becoming increasingly important in our world. We need to be able to handle any event, and we need to be able to recover quickly.

Resilience is a complex concept, and it's not always easy to understand. However, it's a concept that is becoming increasingly important in our world. We need to be able to handle any event, and we need to be able to recover quickly.

Resilience is a complex concept, and it's not always easy to understand. However, it's a concept that is becoming increasingly important in our world. We need to be able to handle any event, and we need to be able to recover quickly.

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 was one of the earlier, simpler, smaller-scale experiments we conducted at Amazon.com that had the most impact. It was the first of the 12 experiments that we conducted at Amazon.com, and it was the first of the 12 experiments that we conducted at Amazon.com.

It was one of the earlier, simpler, smaller-scale experiments we conducted at Amazon.com that had the most impact. It was the first of the 12 experiments that we conducted at Amazon.com, and it was the first of the 12 experiments that we conducted at Amazon.com.

It was one of the earlier, simpler, smaller-scale experiments we conducted at Amazon.com that had the most impact. It was the first of the 12 experiments that we conducted at Amazon.com, and it was the first of the 12 experiments that we conducted at Amazon.com.

It was one of the earlier, simpler, smaller-scale experiments we conducted at Amazon.com that had the most impact. It was the first of the 12 experiments that we conducted at Amazon.com, and it was the first of the 12 experiments that we conducted at Amazon.com.

It was one of the earlier, simpler, smaller-scale experiments we conducted at Amazon.com that had the most impact. It was the first of the 12 experiments that we conducted at Amazon.com, and it was the first of the 12 experiments that we conducted at Amazon.com.

It was one of the earlier, simpler, smaller-scale experiments we conducted at Amazon.com that had the most impact. It was the first of the 12 experiments that we conducted at Amazon.com, and it was the first of the 12 experiments that we conducted at Amazon.com.



Response: Etsy's Substitution Test

Developer runs command that brings down the site

- Grab another engineer who had no involvement in the incident
- Explain the context of the problem
- Fill developer in on the details known by the developer at the time
- Ask what they would do

Developer says they would run the same command almost every time

Identify the reasons for why this seemed the right decision at the time

Some Example Google Issues

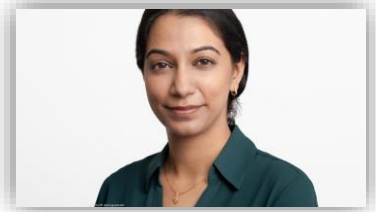
Terminate network in Sao Paulo for testing:

- Hidden dependency takes down links in Mexico which would have remained undiscovered without testing

Turn of data center to find that machines won't come back:

- Ran out of DHCP leases (for IP address allocation) when a large number of machines come back online unexpectedly.

Complexities are introduced as new capabilities are developed. [...] It gets progressively **harder to see where our dependencies are and what might lead to cascading failures.**



ACMQWE WITH JESSICA HARRIS, NICKY KRIVONOS, AND ANITA WANG AND THE LIONEL LINN

At Google, the hidden dependency between our monitoring system and our data center health monitoring is one of the things that could cause the entire line between the data center and the user to go down. This is one of the things that we've discovered and we've learned a lot from it.

THE PROBLEM The problem was that we had a hidden dependency between our monitoring system and our data center health monitoring. This was a hidden dependency because we didn't know it was there until it caused a problem. We didn't know it was there until it caused a problem because we didn't know it was there until it caused a problem.

THE SOLUTION The solution was to discover the hidden dependency and to fix it. We discovered the hidden dependency by looking at the logs and we found it. We found it by looking at the logs and we found it. We found it by looking at the logs and we found it.

THE LESSONS LEARNED The lessons learned from this experience are that we should always be looking for hidden dependencies. We should always be looking for hidden dependencies because we don't know what we don't know.

THE TAKEAWAY The takeaway from this experience is that we should always be looking for hidden dependencies. We should always be looking for hidden dependencies because we don't know what we don't know.

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

Netflix engineers have built a service for simulating and detecting failures in distributed systems with complex dependencies. The service, called Chaos Engineering, allows engineers to inject failures into a system and observe the impact on the system's behavior. This helps them to identify and fix potential issues before they become production problems.



Netflix engineers have built a service for simulating and detecting failures in distributed systems with complex dependencies. The service, called Chaos Engineering, allows engineers to inject failures into a system and observe the impact on the system's behavior. This helps them to identify and fix potential issues before they become production problems.

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

Key metric: **availability**

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



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

Chaos Engineering: The History

Experimentation to **build confidence** around a system to **withstand turbulent conditions** in production

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 network delays so that services react appropriately

Have Chaos Monkey crash development instances, too!

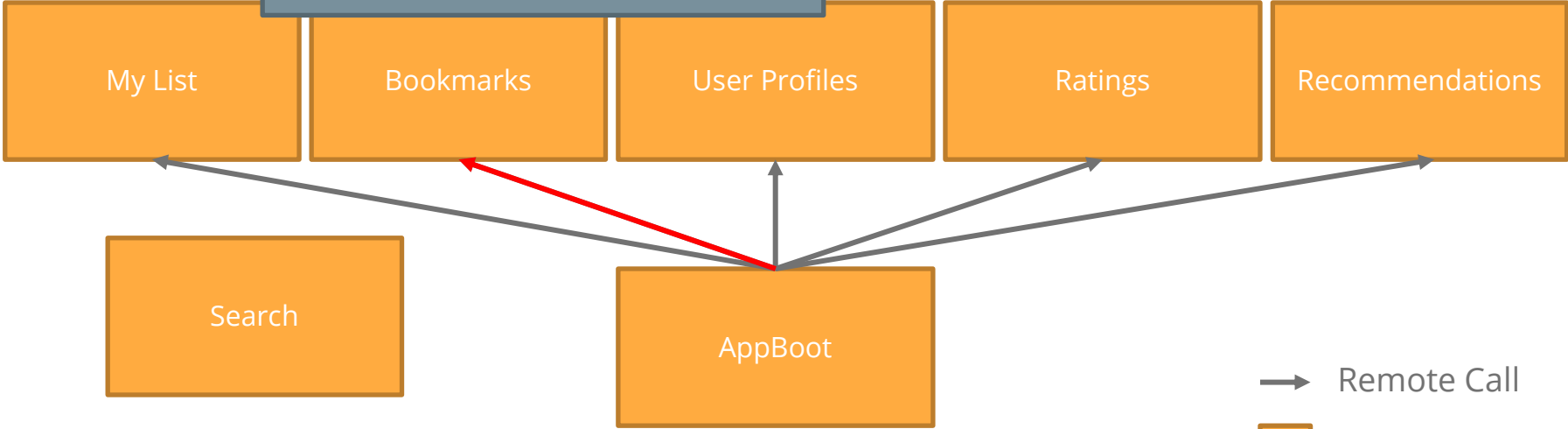




Netflix's Chaos Engineering team... This team is responsible for... chaos engineering... resilience...

Netflix UI: AppBoot

What happens if the bookmark service is down?



➔ Remote Call

▭ Microservice



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, invalidate hypothesis
4. Automate experiments to run continuously

Does everything seem to be **working properly**?

Are users **complaining**?

However, **“works properly”** is too vague a basis for designing experiments.





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?**

...default to starting videos at the beginning rather than providing a "resume from previous location" option.





Many software-based services are implemented as distributed systems with complex behavior and interactions. These systems are often implemented in a way that makes it difficult to understand their behavior in a single or a few experiments.

Steady State Behavior

Back to quality attributes: availability!

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

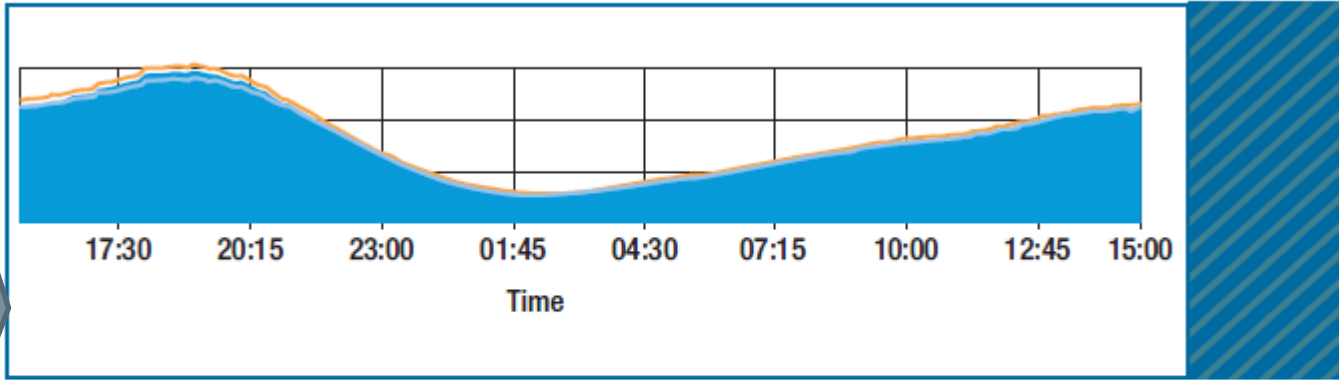


FIGURE 2. A graph of SPS ([stream] starts per second) over a 24-hour period. This metric varies slowly and predictably throughout a day. The orange line shows the trend for the

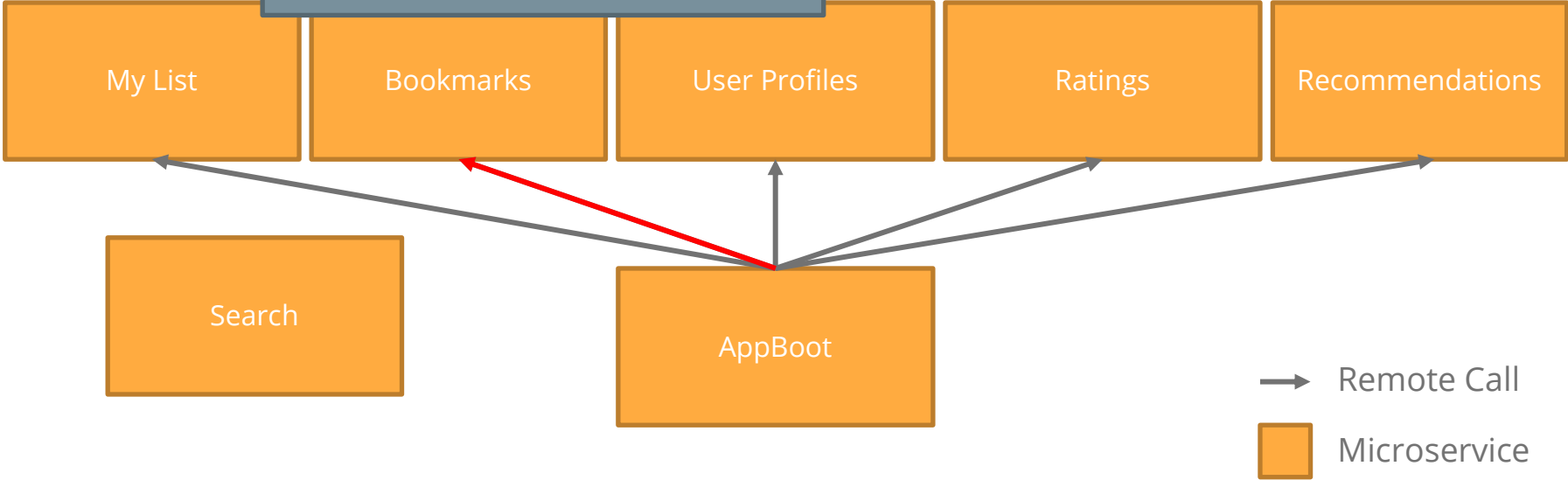
Ultimately, what we care about is whether users can **find content to watch and successfully watch it.**





Netflix UI: AppBoot

What happens if the bookmark service is down?



AppBoot: Bookmarks Down Scenario (Imaginary)

SPS as **core metric**.

Experiment 1:

Outage of bookmarks service causes UI to fail to load, SPS decreases. Code fixed to hide bookmarks if call fails.

Experiment 2:

Outage of bookmarks service hides bookmarks on UI, SPS stays normal.



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?

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



Varying Real-World Events

1. Clients send malformed requests
2. Servers may send malformed responses
3. Servers die
4. Hard disks fill up
5. Memory is exhausted
6. CPU is overloaded
7. Latencies spike
8. **Load from clients can spike**

A recent study reported that 92% of catastrophic system failures **resulted from incorrect handling of nonfatal errors.**



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

Modern software-based services are fundamentally an distributed system with complex dependencies and interactions. These dependencies mean that even small changes can have significant impacts on the system's behavior. This is why chaos engineering is a critical part of building resilient systems.

Chaos engineering is the practice of intentionally introducing failures into a distributed system to build confidence in the system's ability to withstand real-world failures.

Chaos engineering is not a one-time exercise, but a continuous process. It involves running experiments that simulate real-world failures, such as network outages, server crashes, and data center failures. The results of these experiments are used to identify weaknesses in the system and to improve its resilience.

Chaos engineering is a key component of a DevOps culture, where the goal is to build systems that are resilient, reliable, and easy to maintain. It is a practice that is essential for building systems that can withstand the unpredictable nature of the real world.

Chaos Engineering as Continuous Process

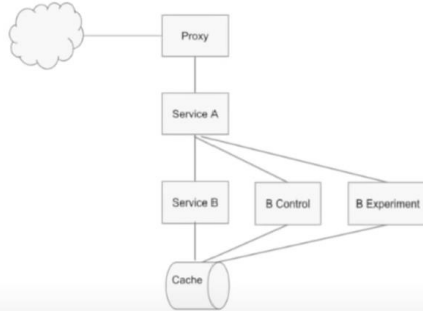
Our system at Netflix changes **continuously**.

Because of these changes, our confidence in past experiments' results **decreases over time**.

Chaos Monkey **runs continuously** during weekdays, and we run Chaos Kong exercises monthly. (2016)



Netflix Today: CHaP



ChAP - How does it work?

NETFLIX



Automatic experimentation
and failure injection with
FIT



Automatic instrumentation
of key performance metrics



Automatic termination
based on key metrics

Automatic experiment design with Monocle

reference: <https://www.youtube.com/watch?v=3WRVgC8SiGc>

Chaos Engineering

By Peter H. Reardon, Head of Product Engineering, LinkedIn; and David S. White, Head of Product Engineering, Amazon

Modern software-based services are implemented as distributed systems with complex dependencies and failure modes. These dependencies make experiments hard to execute and make it difficult to predict the behavior of a system under stress. This article discusses how to design and execute chaos experiments.



REPORT Modern software-based services are implemented as distributed systems with complex dependencies and failure modes. These dependencies make experiments hard to execute and make it difficult to predict the behavior of a system under stress. This article discusses how to design and execute chaos experiments.

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 as 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.