

Chaos Engineering

Chris Timperley · Wednesday, November 4, 2026

Administrivia

- **Homework 5:** you will evolve the API without incurring data loss (expand-contract)
- **Lab:** none this week (Toxiproxy next week)
- **Interested in being a TA next semester?** Drop me a message on Slack!

🎯 Learning Goals

- Define an SLI and SLO for a journey, and calculate its error budget
- Find failure modes from a request's dependencies, rank them with a lightweight FMEA, and state a falsifiable hypothesis
- Design a bounded fault experiment with a control group and an abort condition based on measured signals
- Choose a resilience pattern and predict how a rerun's result should change
- Explain why retried writes can duplicate, and how idempotency keys prevent it

Production is messy, dynamic, and unpredictable



Many clients and devices



Geographically distributed



Unreliable networks



Constantly evolving



Hardware failure



Unpredictable users

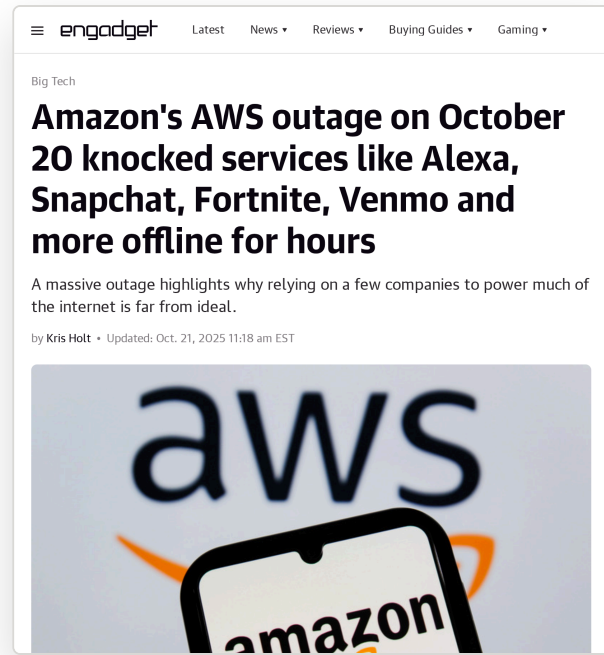
Testing in staging is not enough.
There's no substitute for the real system.

October 20, 2025: An AWS outage spread worldwide

- **DynamoDB**, an AWS database service, **became unreachable** in Northern Virginia
- Services that depended on it failed; full recovery took about **14½ hours**

What would our application do if a dependency became unavailable?

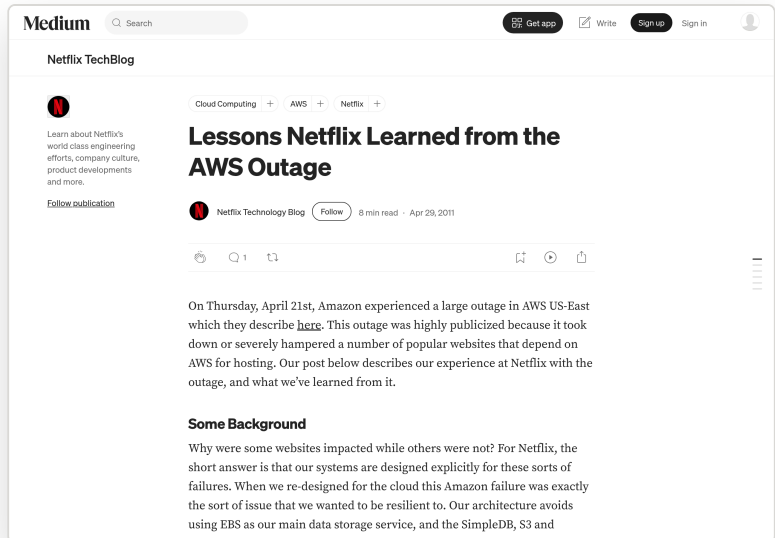
AWS post-event summary ↗



Engadget · October 21, 2025

2011: Netflix had practiced losing servers

- April 2011 AWS outage: Netflix **kept streaming**, with more errors and higher latency
- Before the outage, **Chaos Monkey** already killed production servers to exercise recovery
- Losing an entire **availability zone** required moving traffic manually

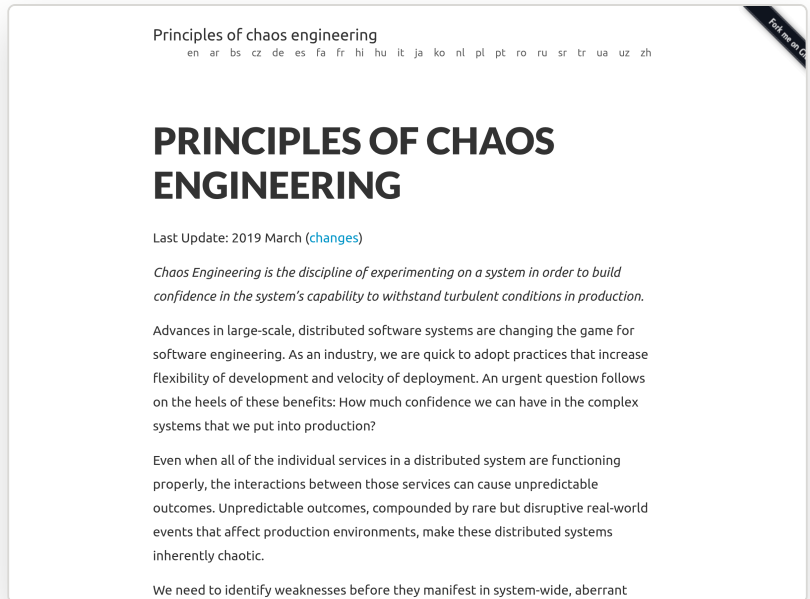


Netflix TechBlog · April 29, 2011

Netflix's lesson: practice losing a whole zone.

Chaos engineering studies deliberate failures

- **Chaos engineering:** experimenting on a system to build confidence that it can withstand turbulent conditions
- Inject a failure and measure whether users can still do what they came to do
- Each experiment tests a **hypothesis** and limits how many users it can affect



principlesofchaos.org

Rare failures let unsafe assumptions accumulate

Chubby is Google's lock service: it helps other services coordinate access to shared resources.



Rare outages
Teams rarely saw Chubby fail

An assumption
They built services as if it could never go down

Wider outages
When Chubby failed, those services failed too

- Google introduced **controlled outages** to expose services that could not tolerate Chubby failing.

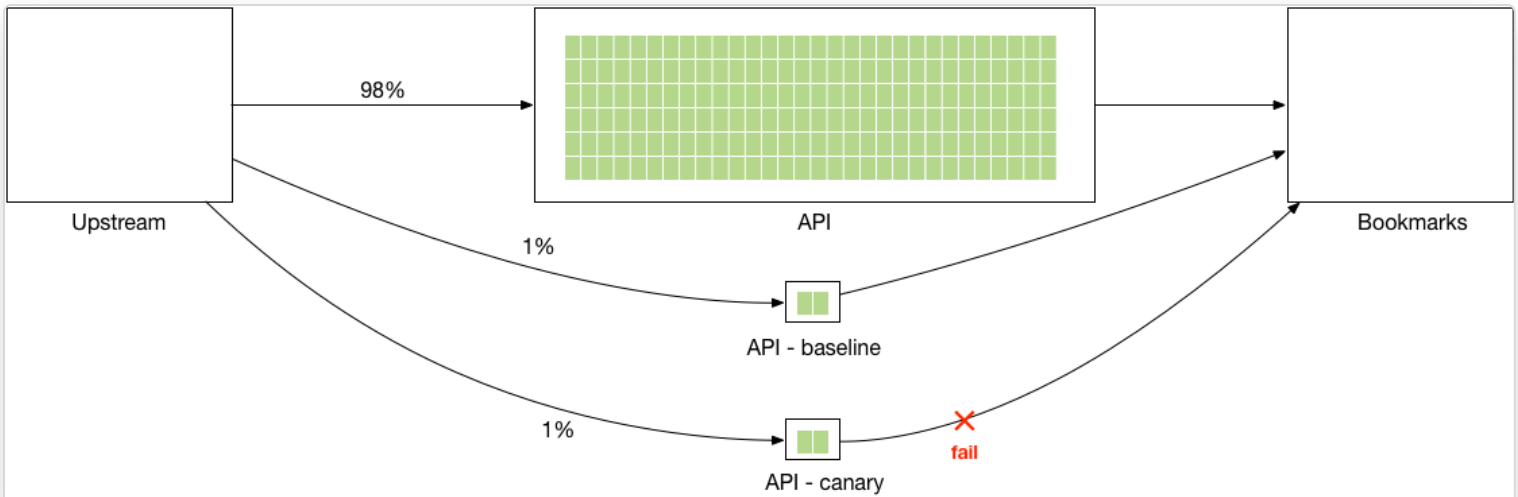
Google SRE Book · The Global Chubby Planned Outage

A chaos experiment has four parts

1. **Steady state**
   A measurable definition of "working"
2. **Hypothesis**
   The steady state survives a specific failure
3. **Run**
   Inject the failure; compare with a control group
4. **Learn**
   Add a defense, then rerun

Adapted from the Principles of Chaos Engineering

Can members still stream when bookmarks fail?



Basiri et al., Automating Chaos Experiments in Production, ICSE-SEIP 2019

- **Hypothesis:** “users should still be able to stream video when the bookmarks service fails”

Steady State

How do we know the system is working?



Bookmarks fail for 1% of members. Did they notice?

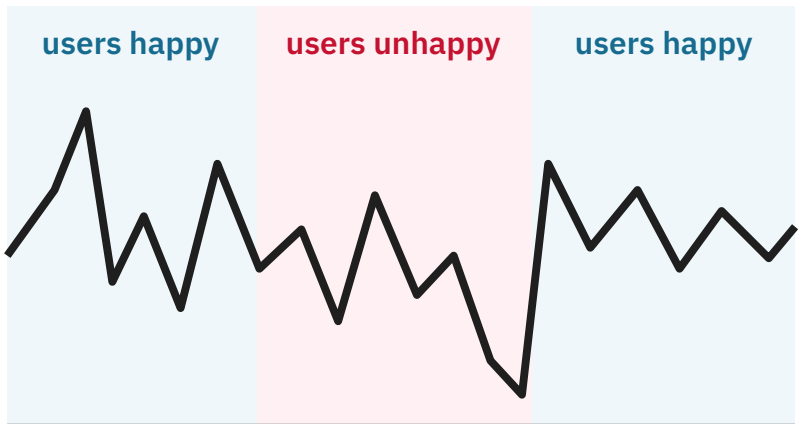
Which signal would tell you?

1. CPU use on the API servers
2. Error messages in the logs
3. Calls to customer service
4. Stream starts per second

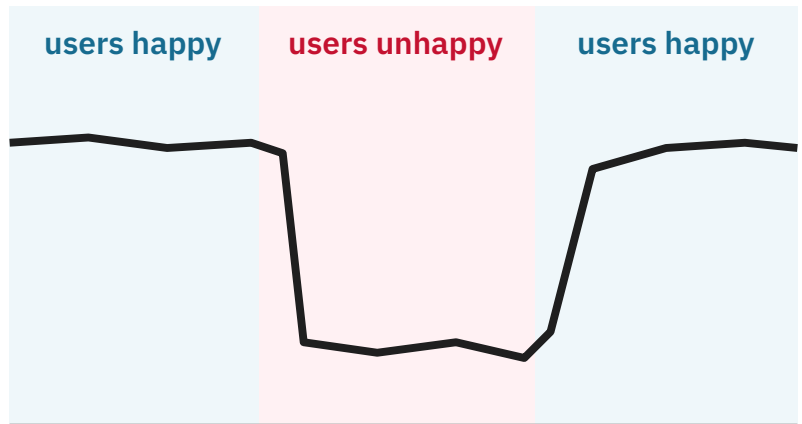


Batelco network operations centre · Masdestructive / Wikimedia Commons (CC BY-SA 3.0)

A useful signal reflects whether users can complete their journey



CPU use

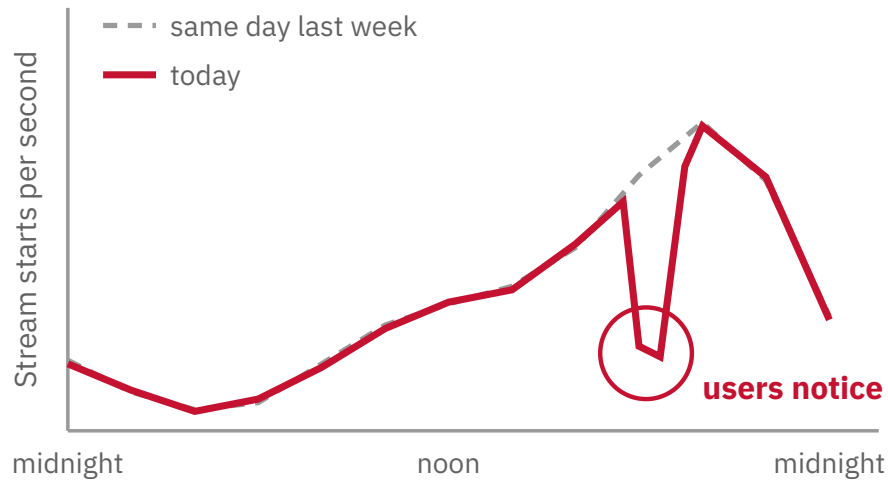


Stream starts per second

Customer-service calls arrive late, and many unhappy members just stop watching

Netflix's steady state is stream starts per second

- **Steady state:** a measurable output that shows the system is behaving normally
- Stream starts per second (SPS) track what members came to do, and follow a predictable daily shape
- An experiment asks whether the steady state **still holds** while something is broken



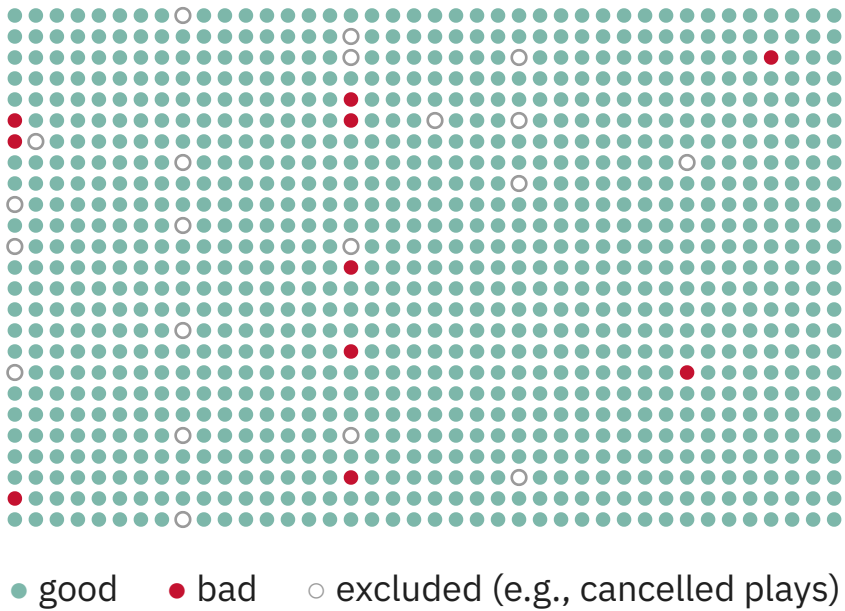
Netflix Tech Blog · SPS: The Pulse of Netflix Streaming

How do we measure reliability?

An SLI is the proportion of valid events that were good

$$\text{SLI} = \frac{\text{good events}}{\text{valid events}}$$

- **Service level indicator (SLI):** this ratio, as a percentage
- Here: 970 good out of 980 valid play requests is **99.0%**
- Unlike p99 latency, a ratio counts every bad event

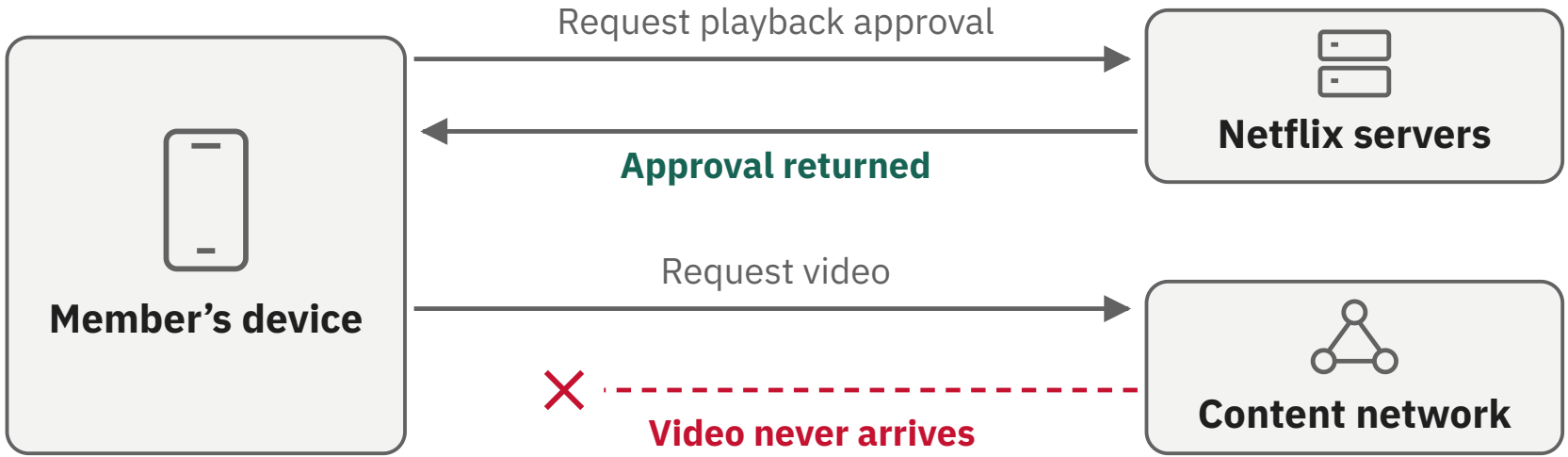


Example: Turn “videos start quickly” into a precise SLI

The proportion of …

Which events count?	play requests from signed-in members, excluding plays the member cancelled,
What is good?	where video is playing within 5 seconds,
Where is it measured?	on the member’s device.

Where you measure determines what you can detect



Server sees: approval succeeded

Device sees: playback failed

Internal services have SLIs too; their users may be other services or teams.

An SLO states how reliable the service needs to be

Service level objective (SLO): a target for an SLI over a stated time window.

Target: at least

99.9%

SLI: proportion of valid plays starting within **5 seconds**

Measured on members' devices

Window: the preceding **28 days**, recalculated at each evaluation

A **service level agreement (SLA)** is a contract that specifies consequences for missing its targets.

Google SRE Workbook · Implementing SLOs

Why not set your SLO to 100%?



Set the SLO below 100%

0

allowed failures

under a 100% target

Failures remain possible
Phones disconnect and devices fail, even with a reliable backend

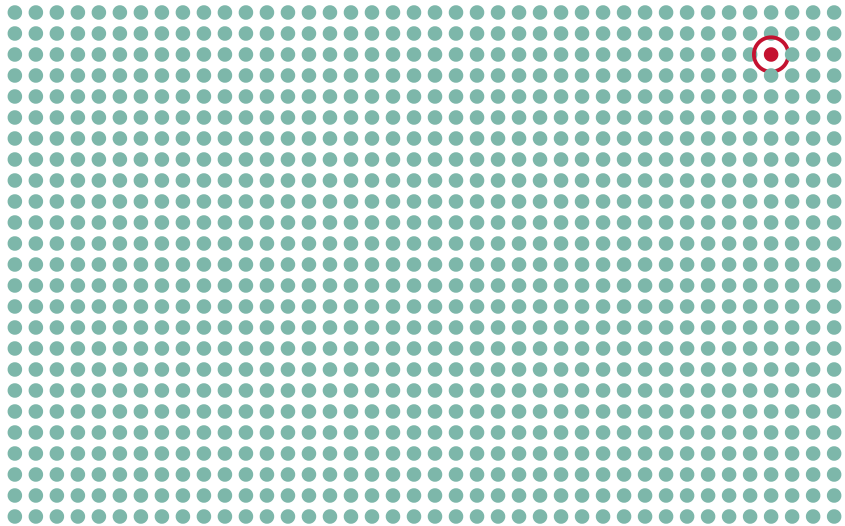
Change carries risk
Releases, security patches, and scaling can cause failures

Extra nines cost more
Device and network failures limit the benefit of backend improvements

Google SRE Workbook · Reliability Targets and Error Budgets

The error budget is the unreliability the SLO allows

- **Error budget:** 1 – SLO, the bad events the team can afford in the window
- 99.9% allows **one bad play start in every 1,000**
- At roughly uniform traffic, about **40 minutes** of complete outage in 28 days



1,000 play starts; the budget allows **one** to be bad

How do we use the error budget?

The error budget decides when to slow down

With **1,000,000 valid plays** in 28 days, a **99.9% SLO** allows **1,000 bad plays**.

Event	Bad plays	Budget left
Start	0	**1,000**
Incident	400	**600**
Release	200	**400**
Experiment	100	**300**

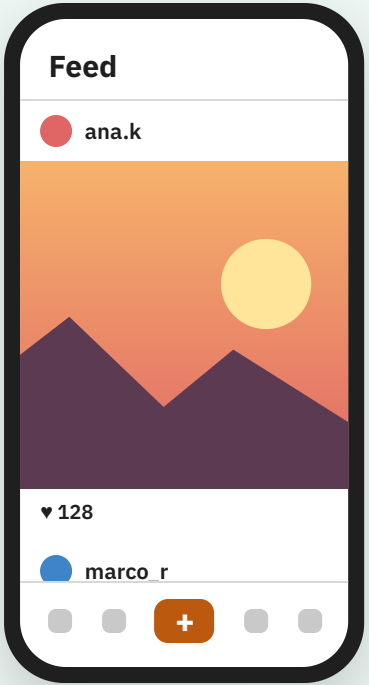
Budget left
Ship features and run experiments within safety limits.

Budget spent
Pause risky changes and fix reliability.

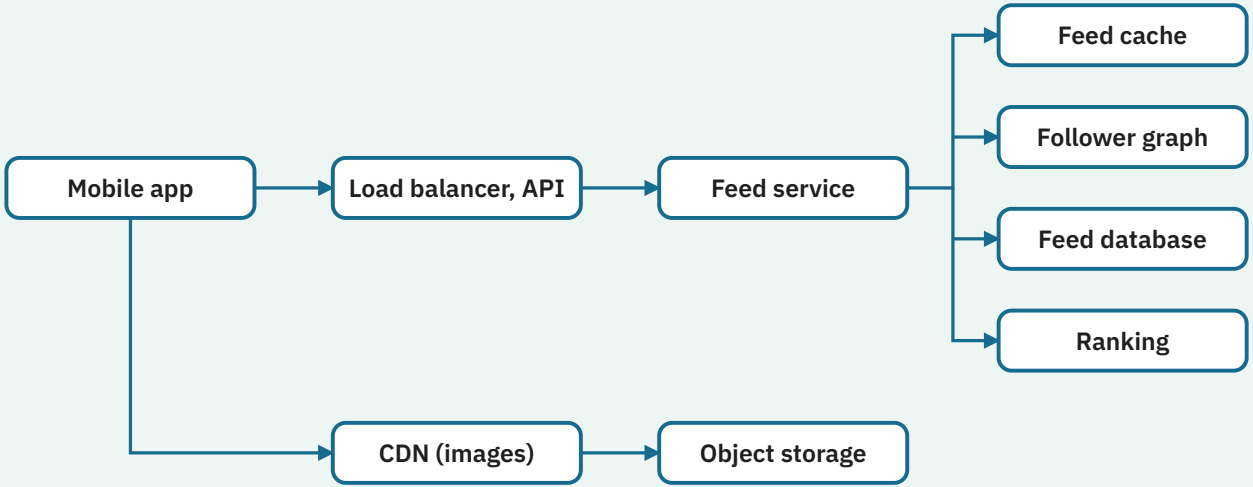
An experiment that degrades playback **spends budget too**.

Product, developers, and operations agree the target and policy in advance.

Your group's system: loading a photo feed



Open the app and see recent photos from the accounts you follow.



Activity: Define the Steady State

- How would you know the feed works for users? Propose two SLIs; pick one
- Set a 28-day SLO for it and work out its error budget. Would last month have met it?

Example: The feed has many valid SLIs

Kind	Proportion of …	Measured at
Latency	Feed loads that succeed within 1 s	Load balancer
Availability	Feed loads that return without an error	Load balancer
Freshness	New posts that reach followers' feeds within a minute	Test poster and follower
Completeness	Feed loads where every photo appears	The app

One choice: latency at the load balancer, 99% over 28 days. Up to 1 million bad loads; last month's 99.48% used about half.

Hypothesis

What could go wrong, and which failure should we test first?

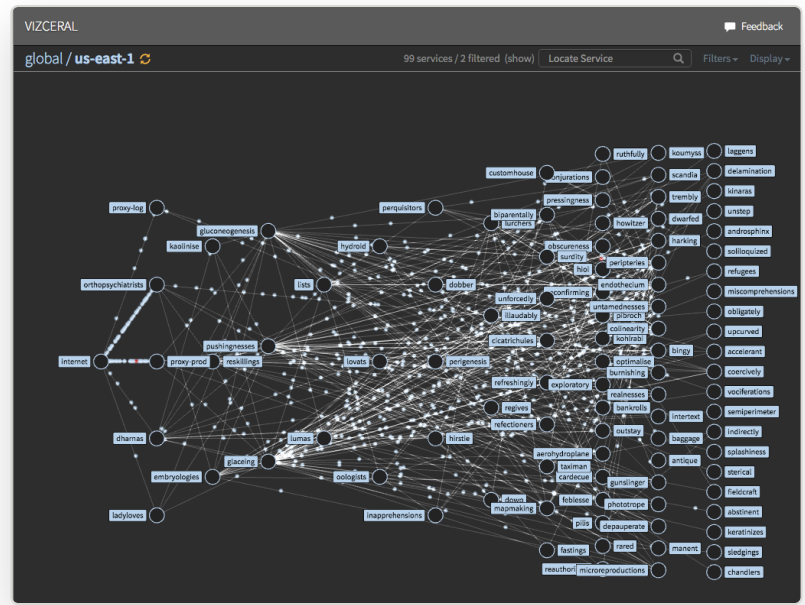
1. Steady state
2. **Hypothesis**
3. Run
4. Learn

One Netflix page depends on hundreds of services



Basiri et al., ICSE-SEIP 2019

- The match score, maturity rating, and video and audio badges each come from a **different service**.



Vizceral, Netflix's traffic map between services (sample data)

How would you decide what to break?

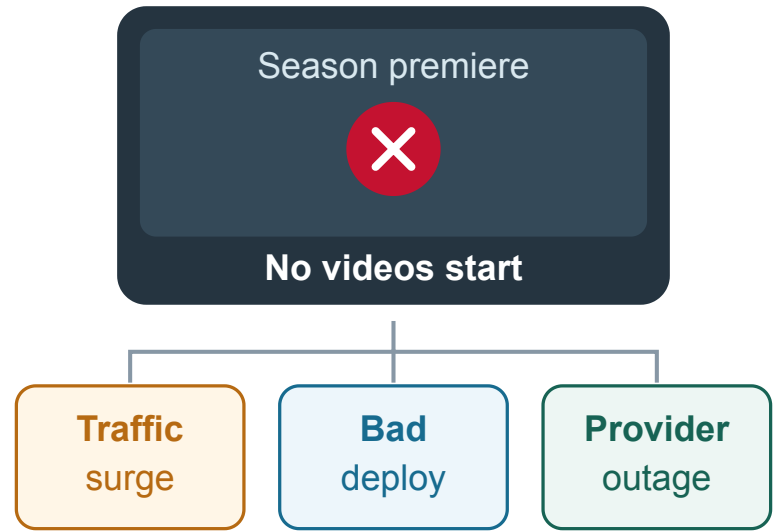


A premortem starts with an imagined outage

Example: The season premiere happened. **Members could not start videos.** What happened?

- Brainstorm possible causes individually
- Collect and discuss them as a team

Failure to test: a traffic surge overwhelms the licensing service.



Gary Klein · Premortem method

Past incidents suggest failures to test

Read incidents and near misses from **your system, its providers, and comparable services**.

- What failed, and how did it spread?
- Could this affect our journey?
- Would the fix survive changes in code or traffic?



Summary of the Amazon DynamoDB Service Disruption in the Northern Virginia (US-EAST-1) Region

We wanted to provide you with some additional information about the service disruption that occurred in the N. Virginia (us-east-1) Region on October 19 and 20, 2025. While the event started at 11:48 PM PDT on October 19 and ended at 2:20 PM PDT on October 20, there were three distinct periods of impact to customer applications. First, between 11:48 PM on October 19 and 2:40 AM on October 20, Amazon DynamoDB experienced increased API error rates in the N. Virginia (us-east-1) Region. Second, between 5:30 AM and 2:09 PM on October 20, Network Load Balancer (NLB) experienced increased connection errors for some load balancers in the N. Virginia (us-east-1) Region. This was caused by health check failures in the NLB fleet, which resulted in increased connection errors on some NLBs. Third, between 2:25 AM and 10:36 AM on October 20, new EC2 instance launches failed and, while instance launches began to succeed from 10:37 AM, some newly launched instances experienced connectivity issues which were resolved by 1:50 PM.

DynamoDB

Between 11:48 PM PDT on October 19 and 2:40 AM PDT on October 20, customers experienced increased Amazon DynamoDB API error rates in the N. Virginia (us-east-1) Region. During this period, customers and other AWS services with dependencies on DynamoDB were unable to establish new connections to the service. The incident was triggered by a latent defect within the service's automated DNS management system that caused endpoint resolution failures for DynamoDB.

Many of the largest AWS services rely extensively on DNS to provide seamless scale, fault isolation and recovery, low latency, and locality. Services like DynamoDB maintain hundreds of thousands of DNS records to operate a very large heterogeneous fleet of load balancers in each Region. Automation is crucial to ensuring that these DNS records are updated frequently to add additional capacity as it becomes available, to correctly handle hardware failures, and to efficiently distribute traffic to optimize customers' experience. This automation has been designed for resilience, allowing the service to recover from a wide variety of operational issues. In addition to providing a public regional endpoint, this automation maintains additional DNS endpoints for several dynamic DynamoDB variants including a FIPS compliant endpoint, an IPv6 endpoint, and account-specific endpoints. The root cause of this issue was a latent race condition in the DynamoDB DNS management system that resulted in an incorrect empty DNS record for the service's regional endpoint (**dynamodb.us-east-1.amazonaws.com**) that the automation failed to repair. To explain this event, we need to share some details about the DynamoDB DNS management architecture. The system is split across two independent components for availability reasons. The first component, the DNS Planner, monitors the health and capacity of the load balancers and periodically creates a new DNS plan for each of the service's endpoints consisting of a set of load balancers and weights. We produce a single regional DNS plan, as this greatly simplifies capacity management and failure mitigation when capacity is shared across multiple endpoints, as is the case with the recently launched IPv6 endpoint and the public regional endpoint. A second component, the DNS Enactor, which is designed to have minimal dependencies to allow for system recovery in any scenario, enacts DNS plans by applying the required changes in the Amazon Route53 service. For resiliency, the DNS Enactor operates redundantly and fully independently in three different Availability Zones (AZs). Each of these independent instances of the DNS Enactor looks for new plans and attempts to update Route53 by replacing the current plan with a new plan using a Route53 transaction, assuring that each endpoint is updated with a consistent plan even when multiple DNS Enactors attempt to update it concurrently. The race condition involves an unlikely interaction between two of the DNS Enactors. Under normal operations, a DNS Enactor picks up the latest plan and begins working through the service endpoints to apply this plan. This process typically completes rapidly and does an effective job of keeping DNS state freshly updated. Before it begins to apply a new plan, the DNS Enactor makes a one-time check that its plan is newer than the previously applied plan. As the DNS Enactor makes its way through the list of endpoints, it is possible to encounter delays as it attempts a transaction and is blocked by another DNS Enactor updating the same endpoint. In these cases, the DNS Enactor will retry each endpoint until the plan is successfully applied to all endpoints. Right before this event started, one DNS Enactor experienced unusually high delays needing to retry its update on several of the DNS endpoints. As it was slowly working through the endpoints, several other

AWS post-event summary · October 2025

Google SRE Workbook · Learning from Failure

FMEA traces component failures to user impact

Failure mode	System response	User impact
Bookmarks is slow **2 s** instead of 20 ms	**API waits** for the response	**Playback starts late** Play-start SLI may drop

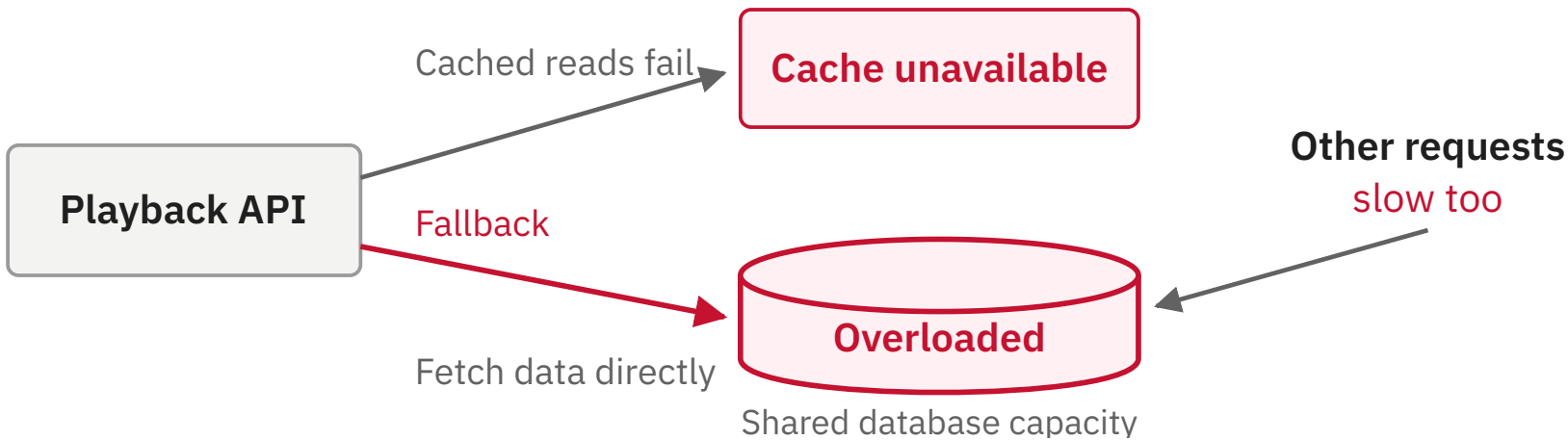
- Failure mode and effects analysis (FMEA) starts with a component failure
- For each component: what if it was unavailable, slow, incorrect, or overloaded?

NASA · Software failure modes and effects analysis

Which failures should we investigate first?

Follow the chain to identify cascading failures

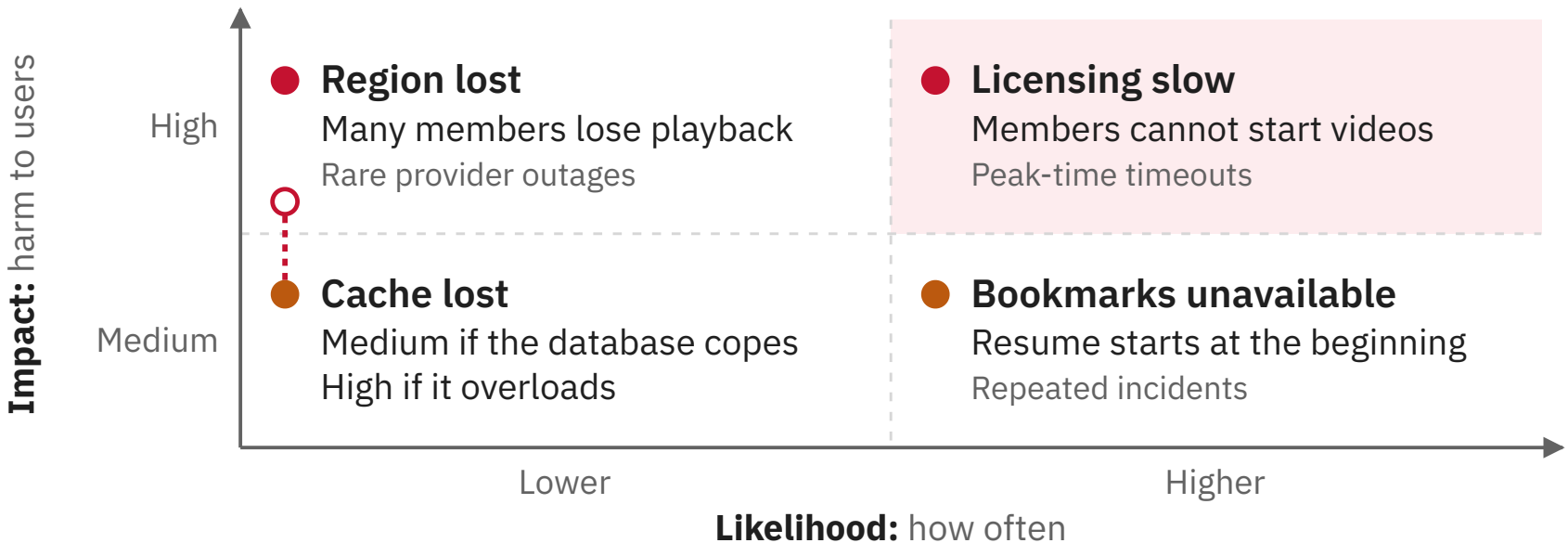
What does each component do in response?



Where do retries, fallbacks, or failover send the work? **What capacity do they share?**

Google SRE · Addressing cascading failures

Prioritize by user impact and likelihood



Experiment on what you can't predict

FROM EVIDENCE

How often does it happen?

- Estimate it from incident history, telemetry, and provider reports
- No recorded incidents can mean **missing evidence**

FROM AN EXPERIMENT

What happens when it does?

- Does the system contain the failure, or does it spread?
- Will losing the cache **overload the database**, or will playback continue?

Start with severe, plausible failures whose effect you can't predict

A hypothesis predicts behavior under failure

Our assumption: playback can proceed without bookmarks.

If every bookmarks call fails for the **experiment group** for **30 minutes**...

> its play-start SLI stays no more than
>
> **0.1 percentage points**
>
> below the control group's.

What result would refute this?

Activity: Find and Rank Failures

- Brainstorm failures: imagine the feed broke last night, recall outages you know, and walk the request hop by hop
- Pick three: what users would see, whether your SLI notices, and their impact and likelihood; circle your top hazard
- Complete the hypothesis on your handout

Example: Test a slow ranking service first

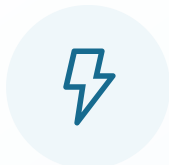
Failure	Users see	SLI notices?	Impact	Likelihood
Ranking responds slowly	**Feeds load slowly**	**Yes**	**H**	**M**
Feed cache is lost	Database overloads; feeds fail	Yes	H	L
CDN fails	Blank photos	No	M	M

Hypothesis: When ranking responds 500 ms slower for the experiment group, the SLI stays within 0.1 percentage points of the control group's.

Running the Experiment

How do we cause a failure and measure its effect safely?

1. Steady state
2. Hypothesis
3. **Run**
4. Learn



KEY CONCEPT

Fault injection

Deliberately introducing a failure at a boundary in the system to observe how it responds.

It is the mechanism behind both **tests**, such as Filibuster on Monday, and **experiments** on a running system.

Code: Failpoints

```
import os

def read_bookmark(member_id):
    if os.getenv("FAIL_BOOKMARK_READ") == "1":
        raise OSError("Injected bookmark read failure")
    return database.read_bookmark(member_id)
```

Mocks replace a dependency; this switch forces an error inside the code to exercise error handling

Service calls: Filibuster and Netflix FIT

Filibuster





Service-Level Fault Injection Testing

Christopher S. Meiklejohn, Carnegie Mellon University, Pittsburgh, PA, United States, cmeiklej@cs.cmu.edu
Andrea Estrada, Carnegie Mellon University, Pittsburgh, PA, United States, arestrad@andrew.cmu.edu
Yiwen Song, Carnegie Mellon University, Pittsburgh, PA, United States, yiwenson@andrew.cmu.edu
Heather Miller, Carnegie Mellon University, Pittsburgh, PA, United States, heather.miller@cs.cmu.edu
Rohan Padhye, Carnegie Mellon University, Pittsburgh, PA, United States, rohanpadhye@cmu.edu

Abstract

Companies today increasingly rely on microservice architectures to deliver service for their large-scale mobile or web applications. However, not all developers working on these applications are distributed systems engineers and therefore do not anticipate *partial failure*: where one or more of the dependencies of their service might be unavailable once deployed into production. Therefore, it is paramount that these issues be raised early and often, ideally in a testing environment or before the code ships to production.

In this paper, we present an approach called *service-level fault injection testing* and a prototype implementation called Filibuster, that can be used to systematically identify resilience issues early in the development of microservice applications. Filibuster combines static analysis and concolic-style execution with a novel dynamic reduction algorithm to extend existing functional test suites to cover failure scenarios with minimal developer effort. To demonstrate the applicability of our tool, we present a corpus of 4 real-world industrial microservice applications containing bugs. These applications and bugs are taken from publicly available information of chaos engineering experiments run by large companies *in production*. We then demonstrate how all of these chaos experiments could have been run during development instead, and the bugs they discovered detected long before they ended up in production.

CCS Concepts: • **Computer systems organization** → **Reliability.**

Keywords: fault tolerance, fault injection, verification

ACM Reference Format:
Christopher S. Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. 2021. Service-Level Fault Injection Testing. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA. ACM, New York, NY, USA,* 15 pages. https://doi.org/10.1145/3472883.3487003

1 Introduction

Nowadays, large-scale web applications with significant user bases are typically built using a microservice architecture. All companies listed in the Fortune 50 are either hiring for, or have publicly discussed, their use of a microservice architecture to deliver service to their users. The recent popularity of microservice architectures is a direct result of the benefits that this architectural style brings to the organizations that adopt it–smaller teams can focus on individual services written in the best programming language to solve their problem, and are thus able to more rapidly deliver software at scale. However, while microservice architectures help reduce the burden of coordinating changes between teams to the same application, they are known to increase software complexity.

The harsh reality of microservices is that they suddenly force every developer to become a cloud/distributed systems engineer, dealing with the complexity that is inherent in distributed systems [38]. Specifically, *partial failure*, where the unavailability of one or more services can adversely impact the system in unknown ways.

This paper presents the *service-level fault injection testing* (SFIT) technique, as well as Filibuster, a corresponding tool for automatically testing microservice applications for resilience issues related to partial failure. Filibuster can extend an existing functional test suite to cover failure scenarios automatically.

To demonstrate the additional complexity a developer faces when moving from a monolithic architecture to a microservice architecture, consider an application that lets you stream audiobooks. One decomposition [43] of this application into components could look like the following, with one component for each user functionality: storage of audio files, storage of audiobook metadata, storage of user permissions, DRM authorization, content delivery service, user

This work is licensed under a Creative Commons Attribution International 4.0 License.
SoCC '21, November 1–4, 2021, Seattle, WA, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8638-8/21/11.
https://doi.org/10.1145/3472883.3487003

Choose which request and which dependency call will fail

Filibuster · Meiklejohn et al. · SoCC 2021

Network: Toxiproxy

README Code of conduct MIT license Security

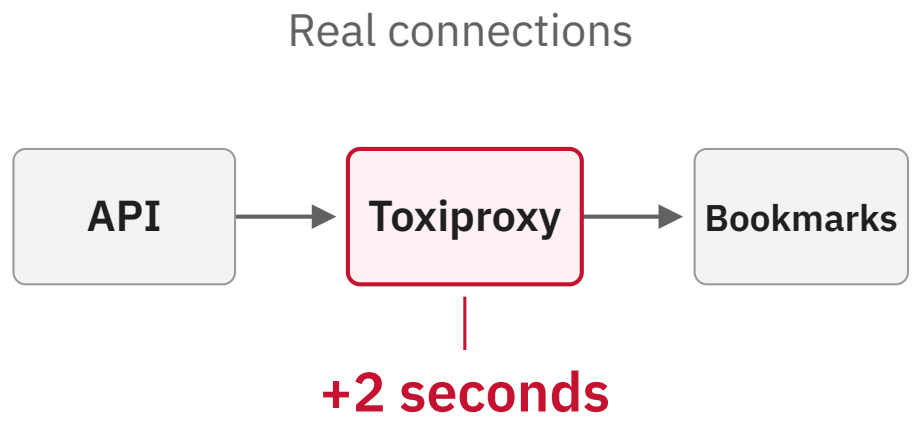
Toxiproxy

release v2.9.0 Test passing



Toxiproxy is a framework for simulating network conditions. It's made specifically to work in testing, CI and development environments, supporting deterministic tampering with connections, but with support for randomized chaos and customization. **Toxiproxy is the tool you need to prove with tests that your application doesn't have single points of failure.** We've been successfully using it in all development and test environments at Shopify since October, 2014. See our blog post on resiliency for more information.

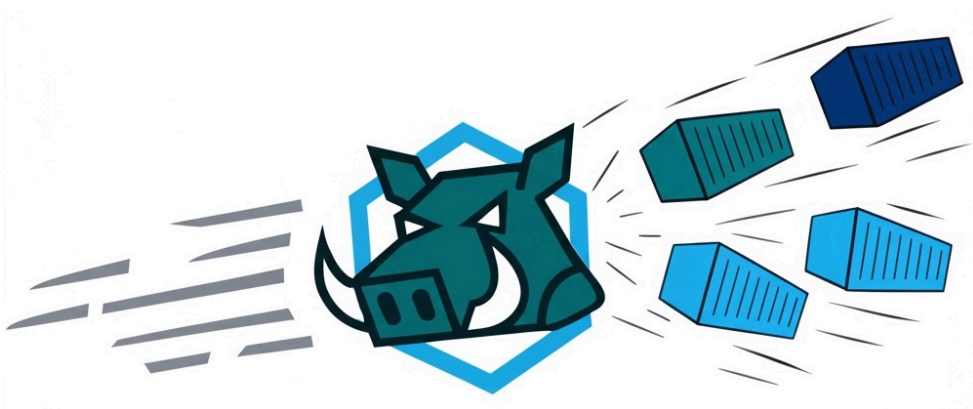
Toxiproxy usage consists of two parts. A TCP proxy written in Go (what this repository contains) and a client communicating with the proxy over HTTP. You configure your application to make all test connections go through Toxiproxy and can then manipulate their health via HTTP. See Usage below on how to set up your project.



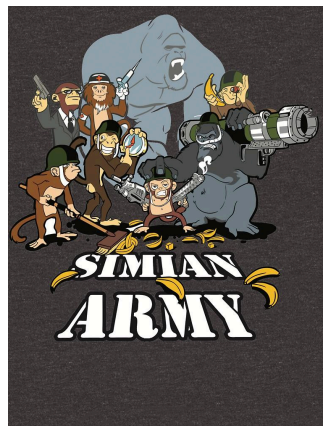
Delay and drop traffic between real services; the client and connection pool wait

Shopify · Toxiproxy

Containers and hosts: Pumba and Chaos Monkey



Pumba
Kill a container



Chaos Monkey
Terminate an instance

A lost replica shifts traffic to survivors; do they have enough capacity?

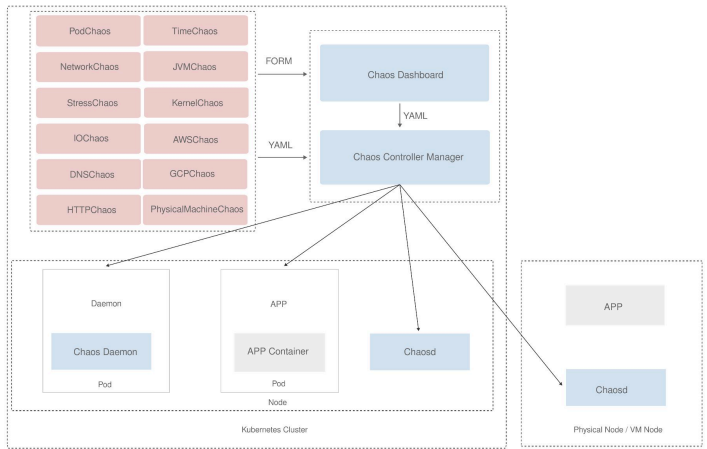
Pumba project · Netflix · Simian Army

Platform orchestration: Chaos Mesh

Fault injection

Fault injection is the key of Chaos experiments. Chaos Mesh covers a full range of faults that might occur in a distributed system, and provides three comprehensive and fine-grained fault types: basic resource faults, platform faults, and application-layer faults.

- Basic resource faults:
  - PodChaos: simulates Pod failures, such as Pod node restart, Pod's persistent unavailablility, and certain container failures in a specific Pod.
  - NetworkChaos: simulates network failures, such as network latency, packet loss, packet disorder, and network partitions.
  - DNSChaos: simulates DNS failures, such as the parsing failure of DNS domain name and the wrong IP address returned.
  - HTTPChaos: simulates HTTP communication failures, such as HTTP communication latency.
  - StressChaos: simulates CPU race or memory race.
  - IOChaos: simulates the I/O failure of an application file, such as I/O delays, read and write failures.
  - TimeChaos: simulates the time jump exception.
  - KernelChaos: simulates kernel failures, such as an exception of the application memory allocation.
- Platform faults:
  - AWSChaos: simulates AWS platform failures, such as the AWS node restart.
  - GCPChaos: simulates GCP platform failures, such as the GCP node restart.
- Application faults:
  - JVMChaos: simulates JVM application failures, such as the function call delay.



Choose the target pods, fault, and duration in one experiment definition

Chaos Mesh · fault types and architecture · CC BY 4.0

Files and system calls: libfiu

```
fiu-run -x -c \
  "enable_random name=posix/io/rw/read,probability=0.05" \
  fortune
```

Give each `read()` a 5% chance of returning an I/O error

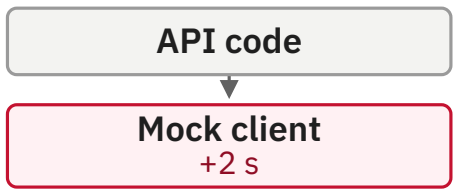
libfiu · documented POSIX example · UnreliableFS · mounted file-system faults

Match the injection mechanism to the failure

Where to inject	Example tools	What changes
Code	Mocks, failpoints	A return value or code path
Service calls	Filibuster, Netflix FIT	Selected calls fail
Network	Toxiproxy	Traffic is delayed or interrupted
Container or host	Pumba, Chaos Monkey	A running instance disappears
Platform orchestration	Chaos Mesh	Faults target selected cluster resources
Files, system calls	UnreliableFS, libfiu	A read or write fails

💉 Inject the fault where the real failure would occur

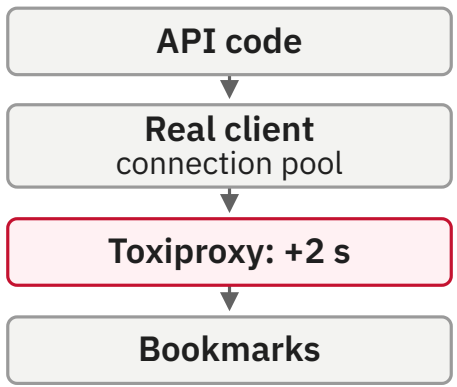
A mock in a test



Real client, pool,
and network bypassed

Checks how API code handles a delayed response

A proxy in the running system



Keeps real connections in use while responses wait

A mock tests code's response to a simulated result. A proxy delay also exercises real connections and resource limits.

How does load change the effect of a failure?

Load testing measures the effect of demand

```
import http from 'k6/http';
import { sleep } from 'k6';

export const options = {
  vus: 20, duration: '5m',
};

export default function () {
  http.get(`${__ENV.BASE_URL}/titles`);
  sleep(1);
}
```

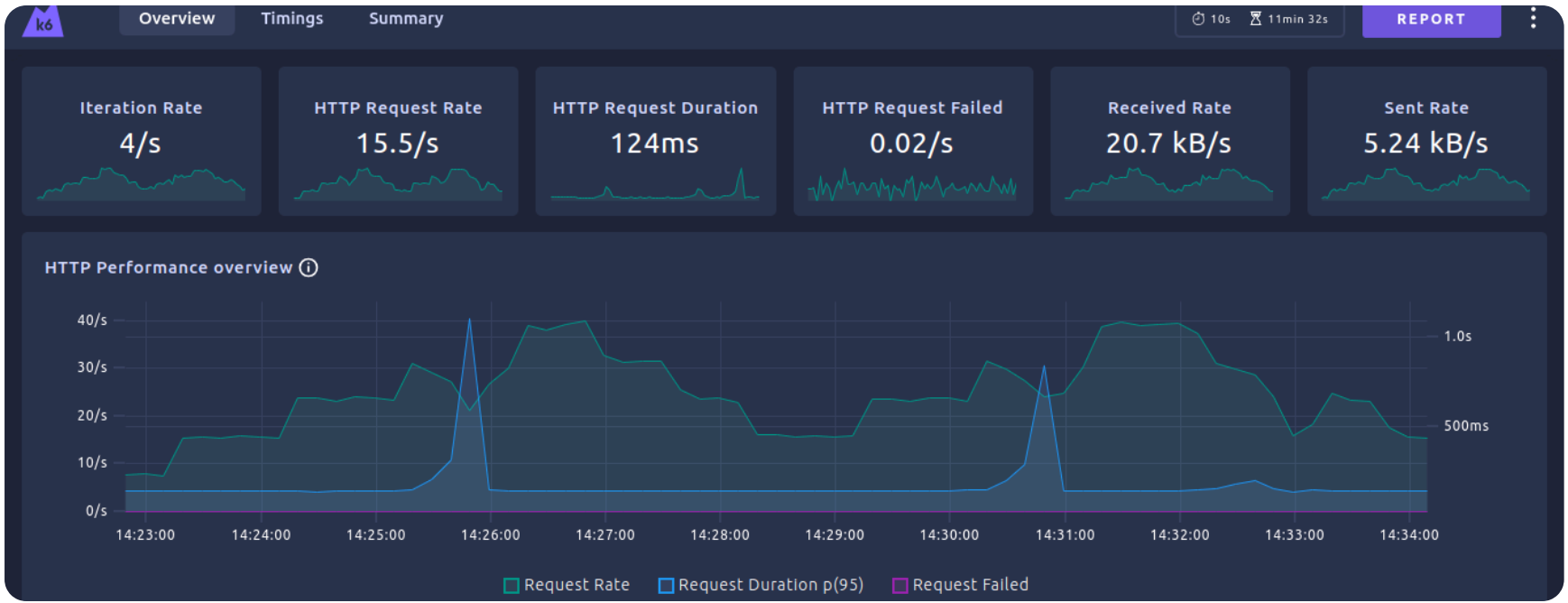


k6: a load generator whose JavaScript script defines what each virtual user does

20 concurrent users repeat the request, then pause for one second

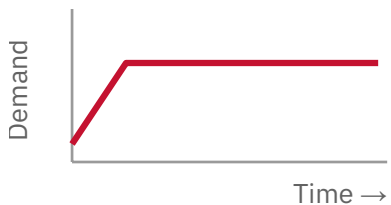
Grafana k6

k6 shows demand and response time together



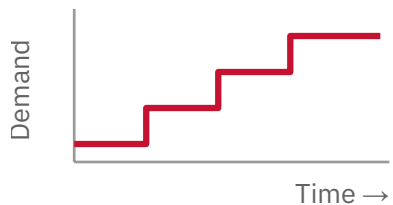
Grafana xk6-dashboard · original example run, overview detail

Each load profile answers a different question



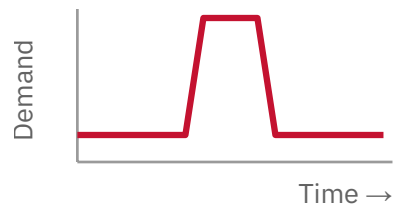
Load

Can it handle expected peak demand?



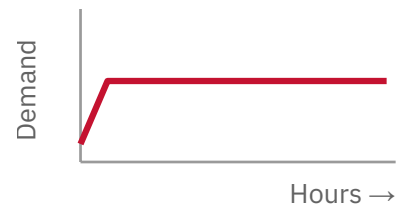
Stress

At what demand does it fail?



Spike

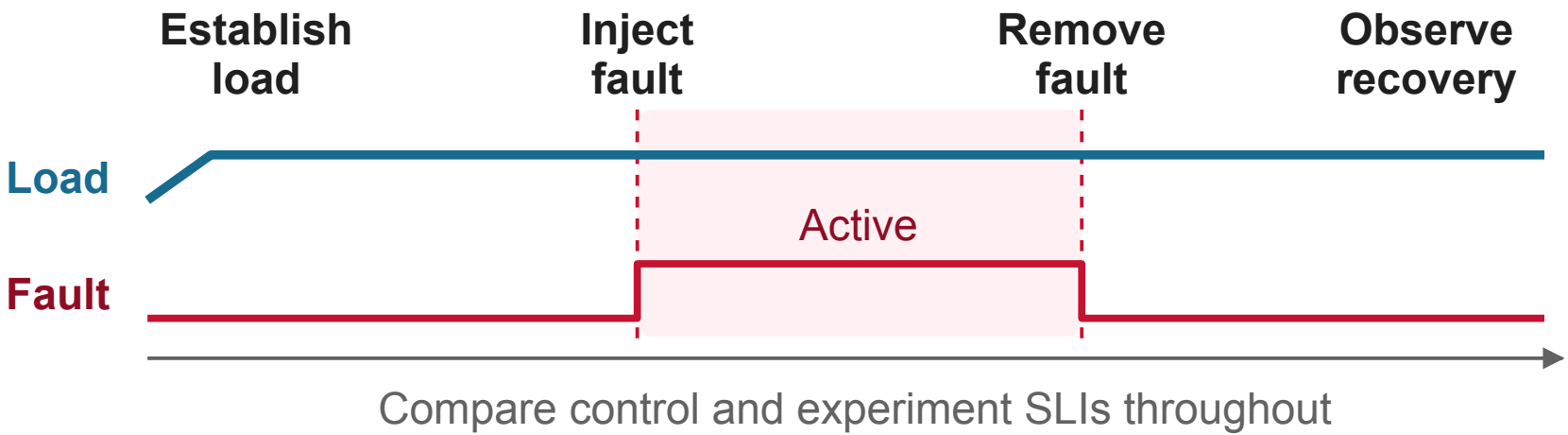
Can it absorb a sudden surge?



Soak

Do memory or connections run out over time?

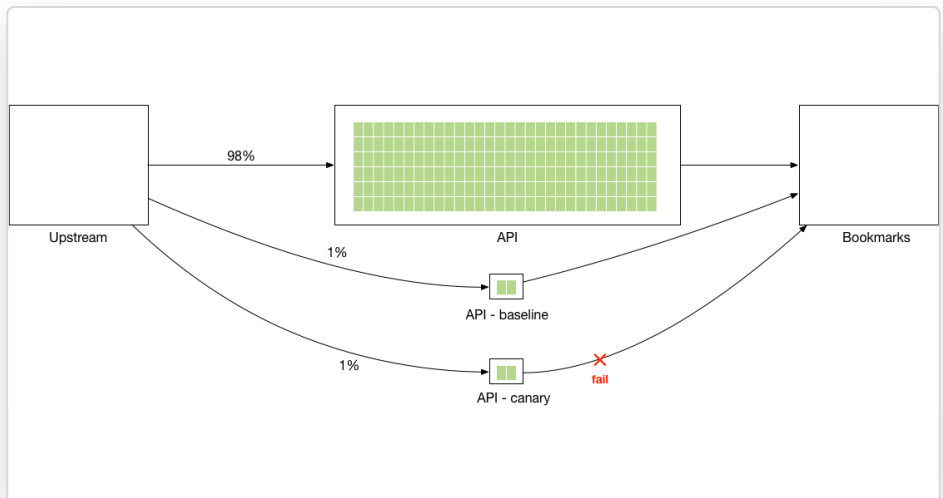
Combine realistic load with the fault



- An idle system can hide failures that appear under load
- **Chaos Kong:** evacuating one region shifts demand to the others

Compare an experiment group with a control group

- Two identical new server groups, each serving about **1% of members**
- Only the **experiment** group's calls to bookmarks fail
- Same moment, same servers: time of day, deploys, and traffic changes affect both groups equally



Basiri et al., ICSE-SEIP 2019

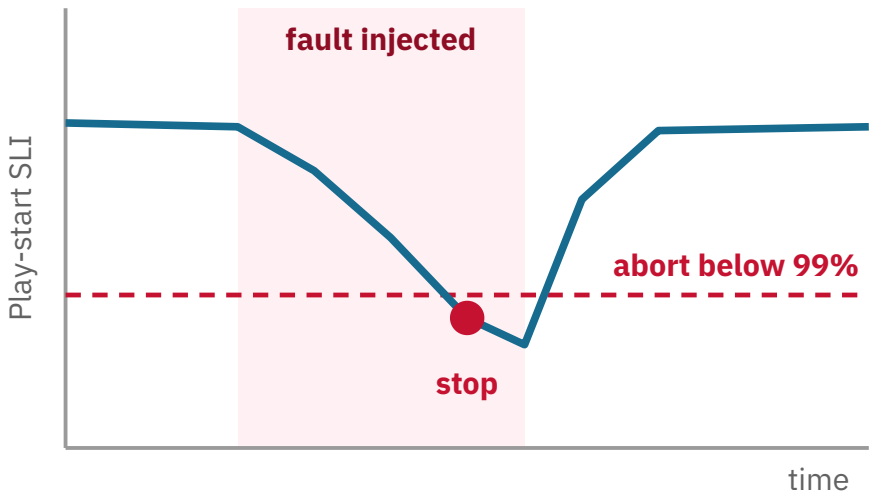
Limit the blast radius so a wrong hypothesis harms few users



Staging	Internal accounts	1% of traffic	Wider
No real users	**Employees** only	**One region**, ten minutes	Only on **evidence** from smaller runs

- **ChAP's limits (Netflix):** about 1% of members per group, at most 5% of a region's traffic in total, weekday business hours, never during a failover

Agree the abort condition before the experiment starts



Abort condition: an observable threshold, agreed in advance, that ends the experiment at once

- Experiment group's SLI below 99% for five minutes
- A shared service nears its limit
- **Name** who stops it
- **Check** that removing the fault is quick

Activity: Plan the Run

For your top hazard, or **ranking responds slowly**:

- Where and how will you inject the fault, and under what load?
- Who is in the experiment and control groups, and how many users?
- When will you stop? Name the signals and their thresholds

Example: Delay ranking for 1% of users

Injection	Toxiproxy adds 500 ms between the feed service and ranking, at peak traffic
Groups	Two random samples of 1% of users in one region, after a run in staging
Abort	Experiment group's SLI below 97% for 5 minutes, or feed-service workers more than 80% busy

Learning from the Result

What do we change when the hypothesis fails?

1. Steady state
2. Hypothesis
3. Run
4. **Learn**

Every result tells the team what to do next



Held

Test a wider scope

The system tolerated this failure. Widen the blast radius carefully, or automate the experiment.



Refuted

Add a defense and rerun

You found a weakness before members did. Choose a defense for this failure mode.



Surprised

Investigate the new failure

Something else broke. Update your understanding of the system before the next run.

How do we improve resilience?



KEY CONCEPT

Resilience pattern

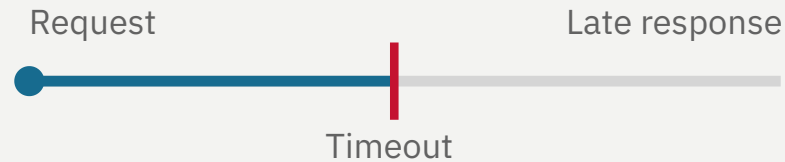
A design element that contains a failure so it does not spread to the rest of the system or to the user.

"Members can still stream" becomes "members can still stream **because** playback starts from the beginning when bookmarks fails". The rerun tests whether that reason holds.

Netflix’s 2011 post named the core defenses

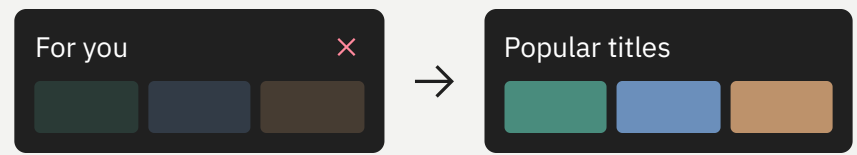
Fail fast

Limit the wait for a slow dependency.



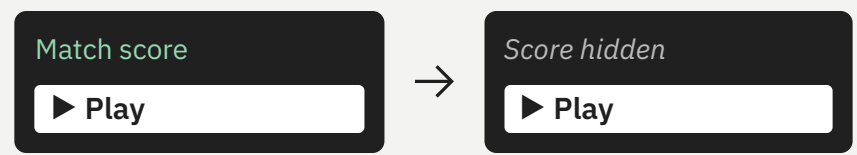
Fallbacks

Use cached or non-personalized results.



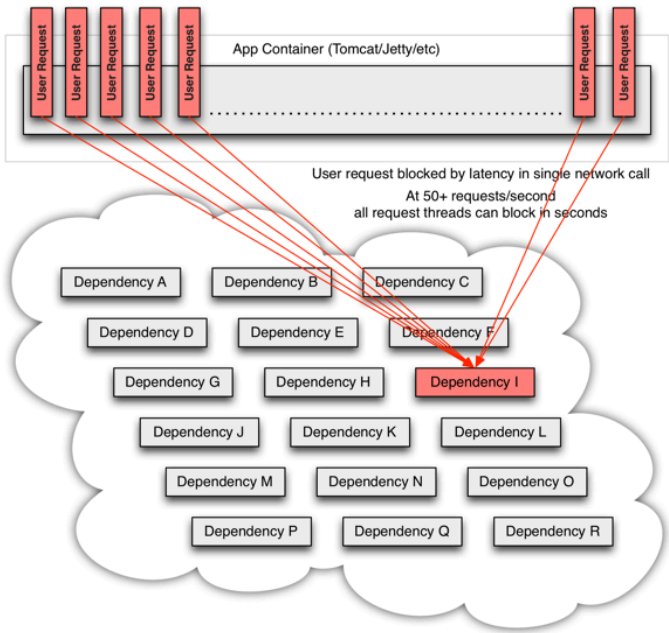
Feature removal

Hide an optional feature when it is slow.



Netflix TechBlog · Lessons Netflix Learned from the AWS Outage (2011)

Timeouts limit the wait for a slow dependency

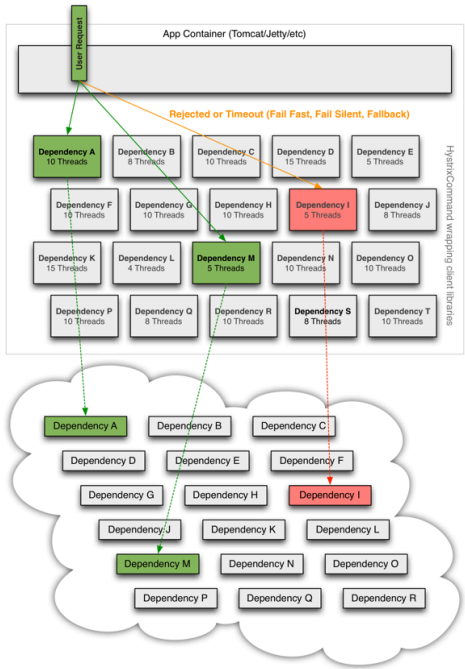


Netflix Hystrix wiki (Apache 2.0)

Timeout: a limit on how long a caller waits for a response.

- Set it **shorter than the caller's own deadline**
- At 50+ requests a second, one slow call can occupy every thread within seconds
- Stop waiting so other requests can use those threads

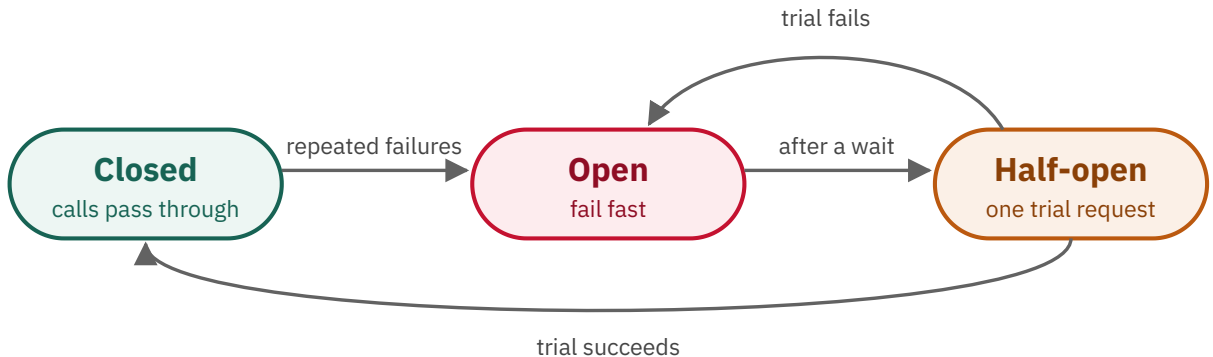
Bulkheads give each dependency its own threads



Netflix Hystrix wiki (Apache 2.0)

- **Bulkhead:** a separate, limited pool of threads or connections for each dependency
- The slow dependency uses up only its own 5 threads; further calls are rejected at once and fall back
- Every other dependency keeps its threads

A circuit breaker stops calls to a failing service



Pseudocode · playback asks the breaker for a bookmark

```
try:
    position = bookmarks_breaker.call(get_bookmark, member, video)
except (CircuitOpen, Timeout, DependencyError):
    position = 0  # start from the beginning
```

Keep the core journey working with less

DEPENDENCY FAILURE

Graceful degradation

Reduce service when a dependency fails.

Ratings fail
Hide the match score.

Personalization fails
Show popular titles.

Bookmarks fails
Start from the beginning.

OVERLOAD

Load shedding

Reject low-priority work under overload.

Playback
Protect the core journey.

Prefetching
Drop before playback.

Logging
Drop first in this example.

Netflix · Prioritized load shedding

Retry with backoff and jitter, and cap the attempts

Retries from many clients over time

Immediate retries



Exponential backoff with jitter

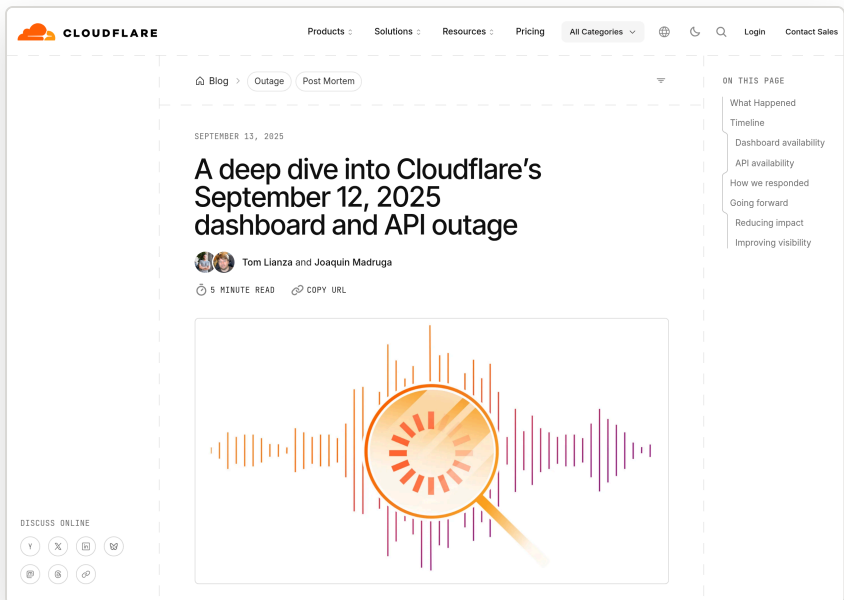


Retry: repeating a failed request in the hope that the fault was brief

Retries multiply across layers

Device	**3**
API	**× 3**
Mid-tier service	**× 3**
Requests to bookmarks	**27**

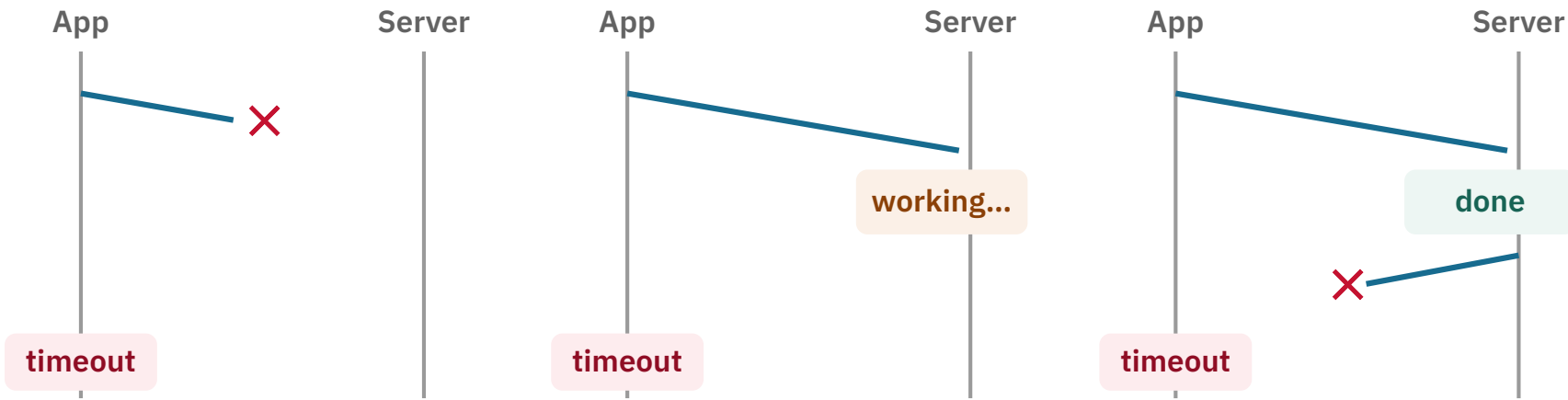
⚠ Unbounded re-requests can overwhelm a shared service



Cloudflare Blog · September 13, 2025

- A dashboard bug **re-sent requests** to the Tenant Service, which authorizes dashboard and API requests
- It was being redeployed; the extra load **overwhelmed** it
- A thundering herd slowed recovery
- A fix: **randomized delays** on dashboard retries

⚠ A timeout does not tell the app whether the request happened



All three look the same to the app: it can only give up or try again



KEY CONCEPT

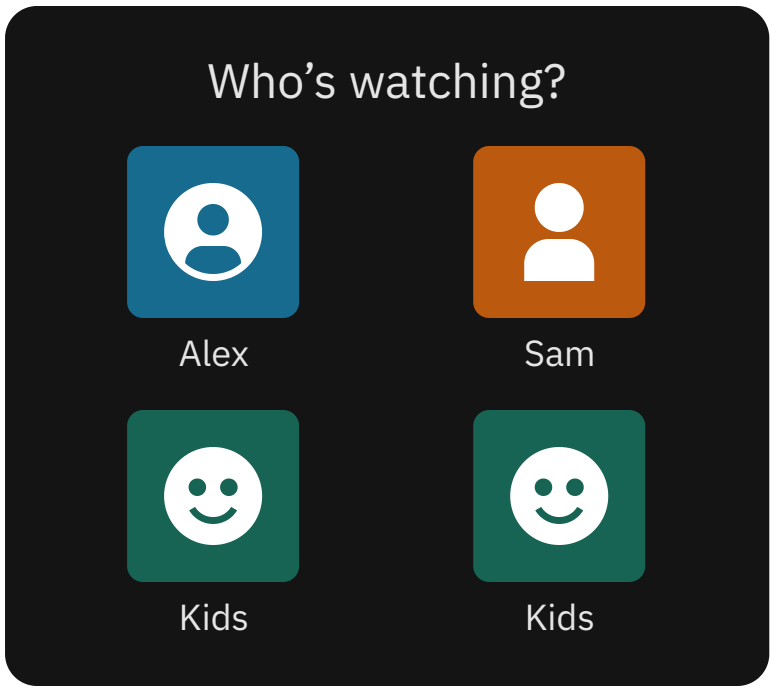
Idempotent operation

Performing it more than once has the same effect as performing it once.

HTTP defines `GET`, `PUT`, and `DELETE` as idempotent and `POST` as not. Beware: the code might break that contract.

RFC 9110 · Section 9.2.2, Idempotent Methods

Example: A retried request adds the profile twice



1. The app sends `POST /profiles` to add “Kids”
2. The server creates the profile
3. The response is lost; the app times out
4. The app retries, and the server creates it **again**

Use an idempotency key to make writes safe to retry

Idempotency key: one unique value for one intended operation, reused on every attempt.

First attempt

```
POST /profiles HTTP/1.1
Idempotency-Key: 4f9b2c1e-77d0

{"name": "Kids"}
```

Profile created; response lost.

Retry · same key

```
POST /profiles HTTP/1.1
Idempotency-Key: 4f9b2c1e-77d0

{"name": "Kids"}
```

The server recognizes the operation.

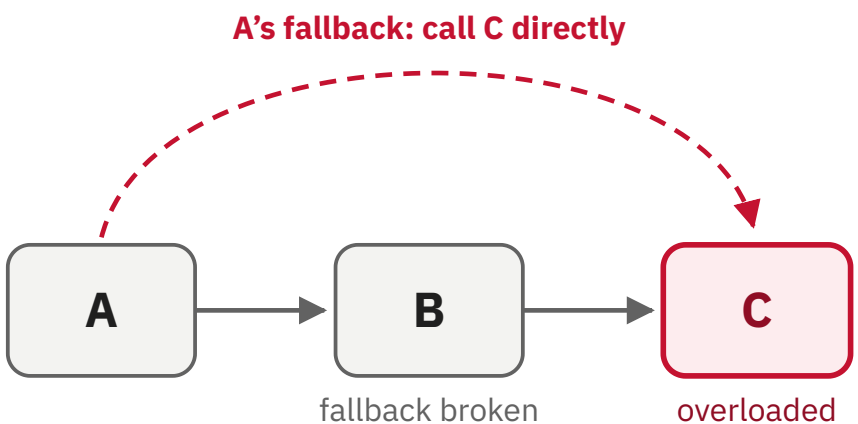
Return the earlier outcome. Create no second profile.

Stripe · Designing robust and predictable APIs with idempotency

Failure Mode	Pattern
A dependency is slow	Timeout
A brief fault	Retry with backoff, **idempotent operations only**
A persistent failure	Circuit breaker
One dependency uses up shared capacity	Bulkhead
Overload	Load shedding
An optional dependency fails	Graceful degradation

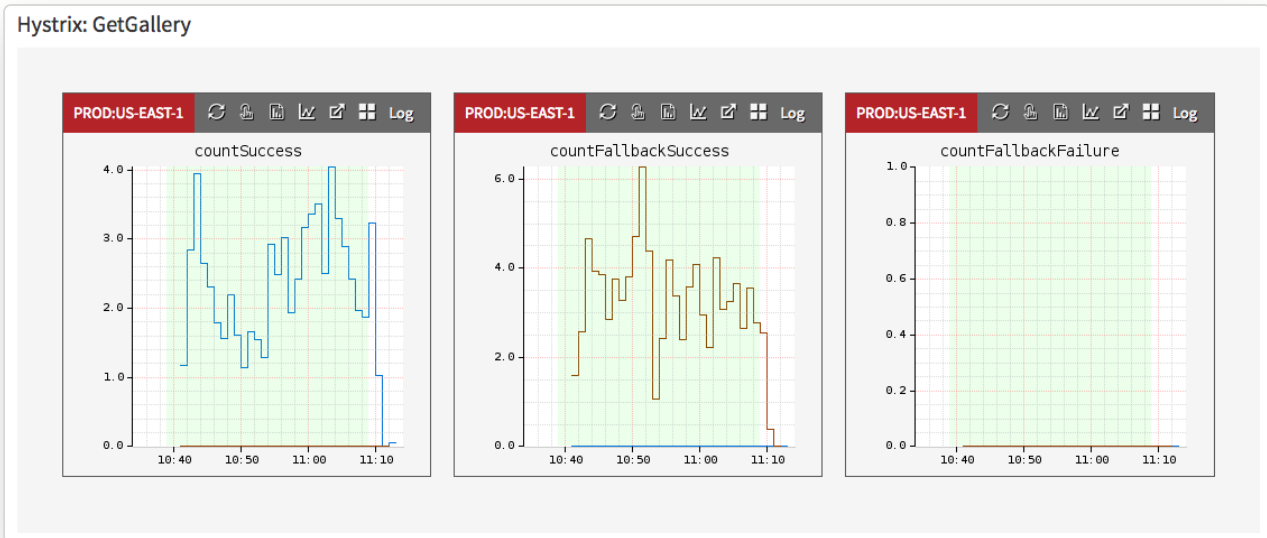
- **Patterns combine:** a timeout alone turns slowness into errors unless something replaces the result

⚠ A defense can make an outage worse



- C was overloaded, and B's own fallback was broken
- A's fallback called C directly, **adding load** to the service that was already struggling
- Rerun with the same failure, groups, and abort condition, and look for **new effects**

A passing rerun shows the defense doing its job

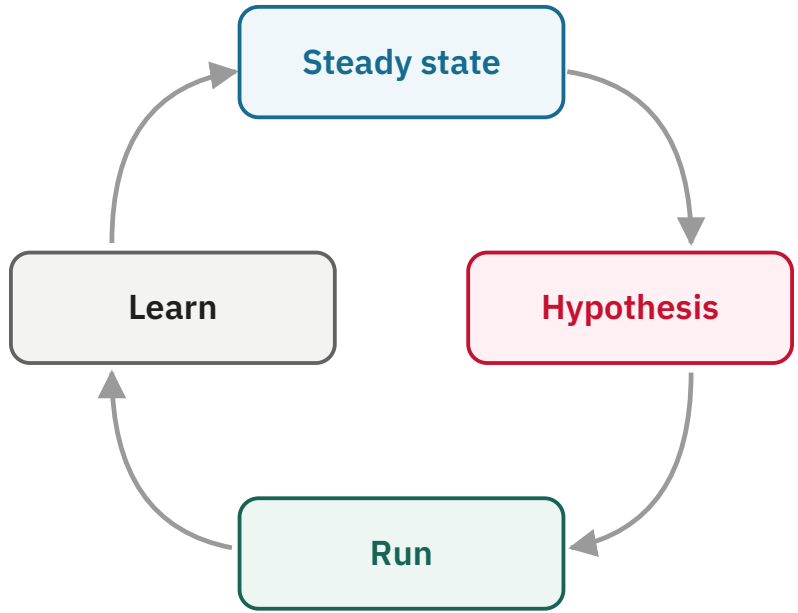


Basiri et al., A Platform for Automating Chaos Experiments, 2016

While Gallery calls failed, **every fallback succeeded** and none failed. A pass shows resilience to **this failure, at this scope and load**, not to every failure.

Automate experiments as the system changes

- Code, load, and dependencies change every day, so a passing experiment goes stale
- Netflix's **Monocle** generates an experiment for each dependency and runs them by priority
- Each run either renews confidence or finds the next weakness



Activity: Choose a Defense

Your hypothesis was refuted: the SLI fell during the experiment.

- Choose a pattern that should contain your hazard: what will users see instead?
- What must the rerun show for you to accept the change?

Example: Fall back to the newest posts

Pattern	A 300 ms timeout on ranking, with graceful degradation: show posts newest first
Users see	An unranked feed instead of a slow one
Rerun	With the same fault, the experiment group's SLI stays within 0.1 percentage points of the control group's, and fallbacks rise

🚀 Today, we learned how to ...

- Define an SLI and SLO for a journey, and calculate its error budget
- Find failure modes from a request's dependencies, rank them with a lightweight FMEA, and state a falsifiable hypothesis
- Design a bounded fault experiment with a control group and an abort condition based on measured signals
- Choose a resilience pattern and predict how a rerun's result should change
- Explain why retried writes can duplicate, and how idempotency keys prevent it