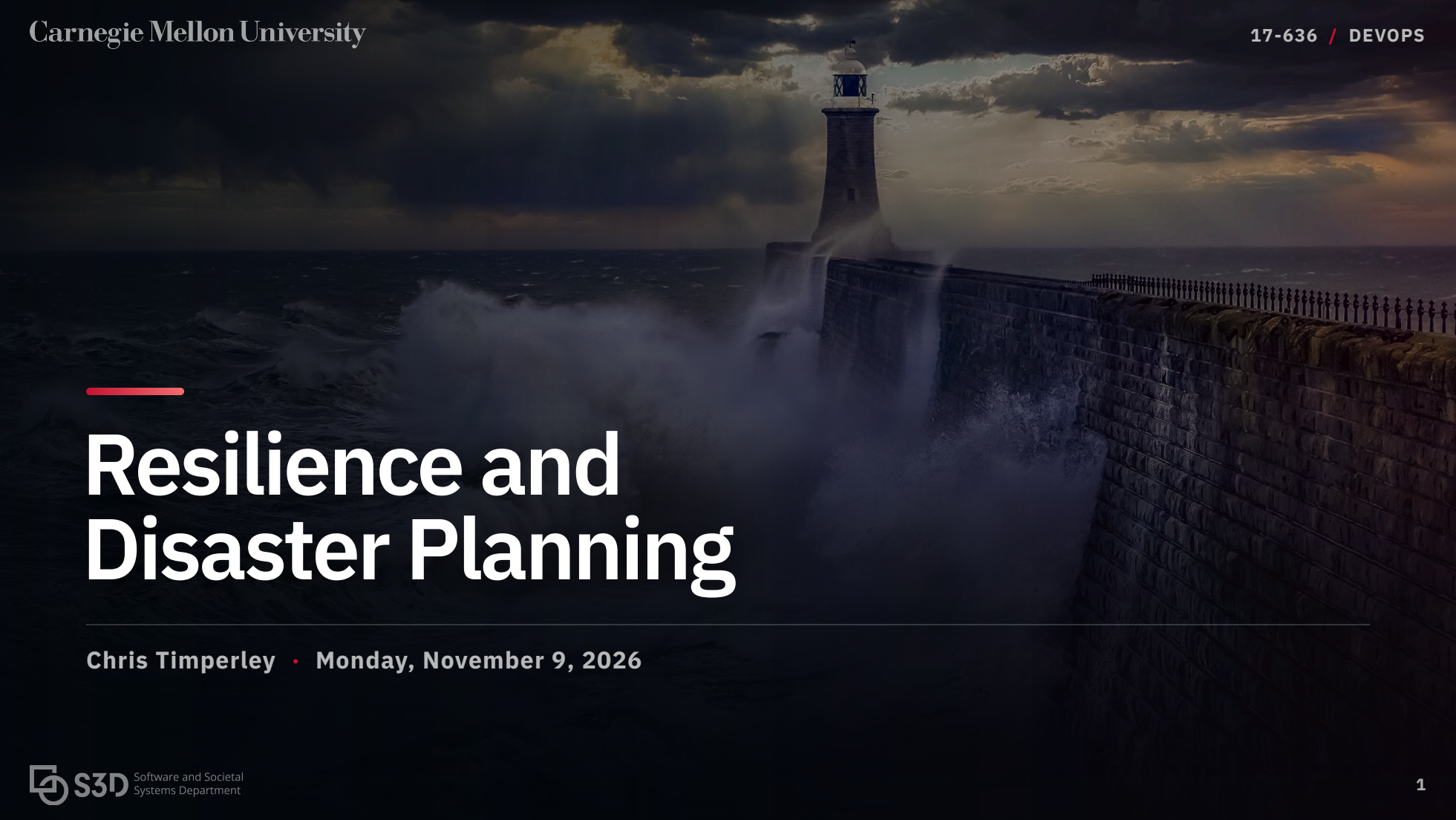


Resilience and Disaster Planning

Chris Timperley · Monday, November 9, 2026

Administrivia

- **Homework 4**: You should **apply** the candidate change
  - this will be necessary for subsequent simulated traffic
- **Lab:** Inject faults via Toxiproxy
  - you will use Toxiproxy and other fault injection tools for HW6
- **Homework 5:** first checkpoint on **Tuesday**; final submission on **Sunday**

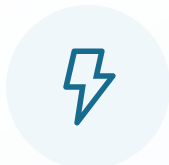
🎯 Learning Goals

- Design a bounded fault experiment with an abort condition
- Choose a resilience pattern and predict how a rerun should change
- Explain retry storms, and how an idempotency key makes retries safe
- Tell an absorbable failure from a disaster, and rank disaster risks
- Explain RPO and RTO, and why only a tested restore protects data
- Judge whether a runbook is safe to follow

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.

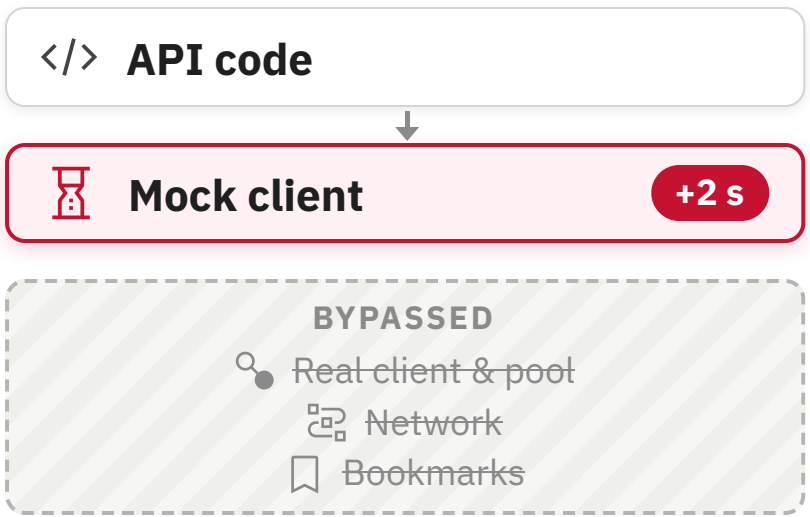
Tests mostly inject in code or service calls, as Filibuster did on Wednesday. **Experiments** inject through a proxy, a container, or a host in the running system.

Faults can be injected at every layer, from code to host

Code	A return value or code path	</> Mocks, Failpoints
Service calls	Chosen calls fail	Filibuster, Netflix FIT
Network	Traffic is delayed or dropped	Toxiproxy, Lab 8
Container or host	An instance disappears	Pumba, Chaos Monkey
Platform	Faults across a cluster	Chaos Mesh
Files, system calls	A read or write fails	libfiu, UnreliableFS

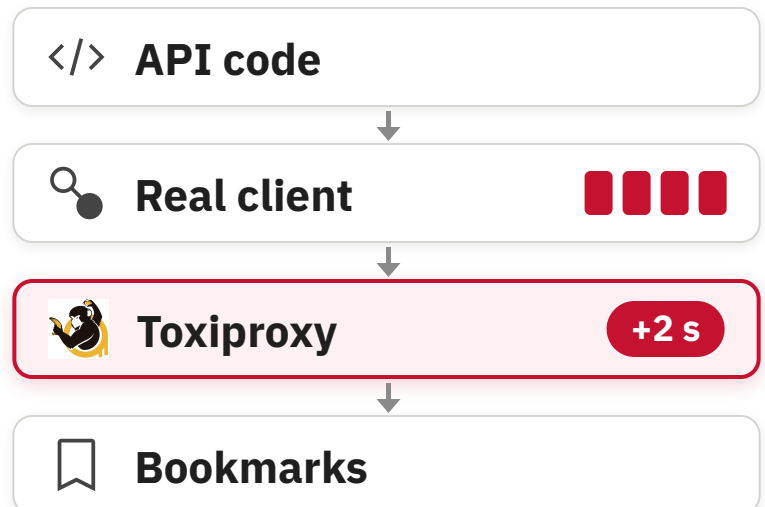
Inject the fault at the layer where the real failure would occur

A mock in a test



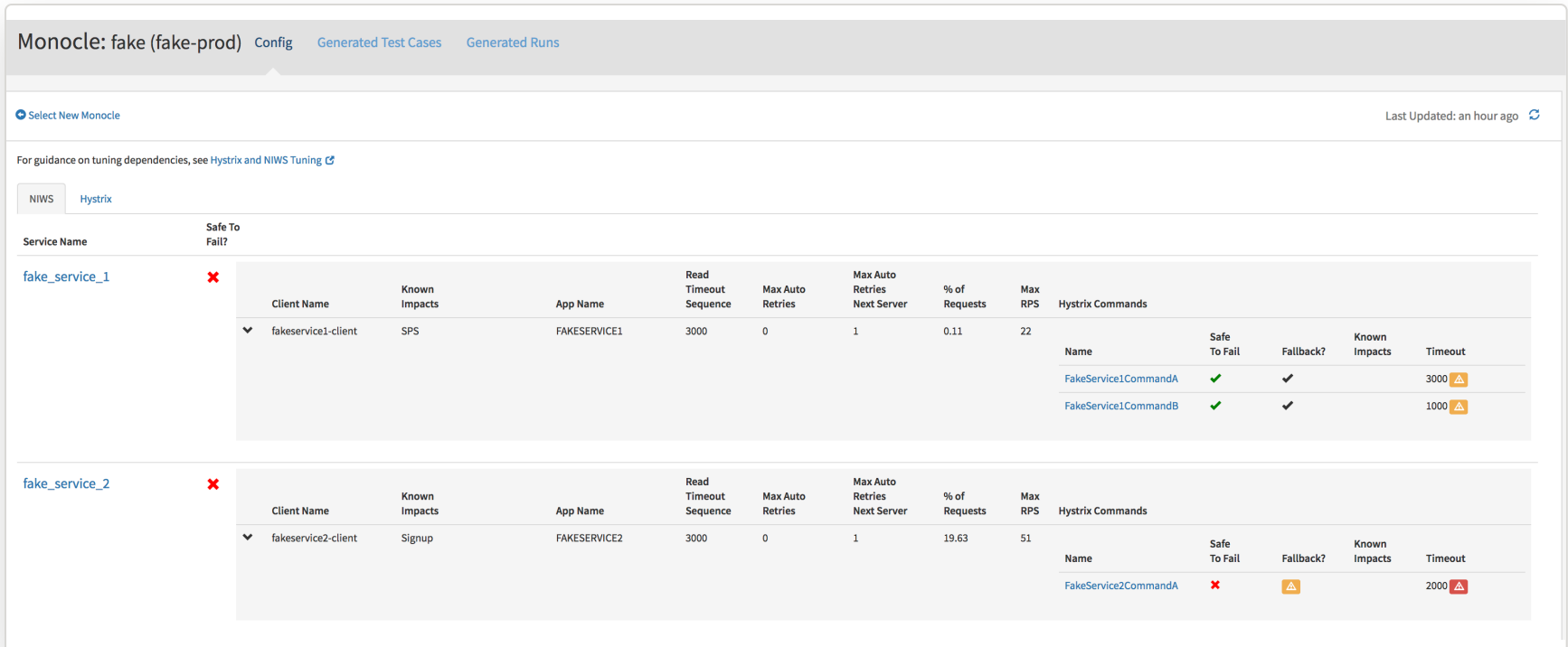
Tests how the code handles a late response

A proxy in the running system



Also shows connections piling up

ChAP generates experiments for each dependency



Monocle: fake (fake-prod) Config Generated Test Cases Generated Runs

Select New Monocle — Last Updated: an hour ago

For guidance on tuning dependencies, see Hystrix and NIWS Tuning

NIWS | Hystrix

Service Name	Safe To Fail?	Client Name	Known Impacts	App Name	Read Timeout Sequence	Max Auto Retries	Max Auto Retries Next Server	% of Requests	Max RPS	Hystrix Commands
fake_service_1	✖	fakeservice1-client	SPS	FAKESERVICE1	3000	0	1	0.11	22	

Name	Safe To Fail	Fallback?	Known Impacts	Timeout
FakeService1CommandA	✔	✔		3000
FakeService1CommandB	✔	✔		1000

Service Name	Safe To Fail?	Client Name	Known Impacts	App Name	Read Timeout Sequence	Max Auto Retries	Max Auto Retries Next Server	% of Requests	Max RPS	Hystrix Commands
fake_service_2	✖	fakeservice2-client	Signup	FAKESERVICE2	3000	0	1	19.63	51	

Name	Safe To Fail	Fallback?	Known Impacts	Timeout
FakeService2CommandA	✖	⚠		2000

ChAP's Monocle view of one service's dependencies (Basiri et al., ICSE-SEIP 2019, Fig. 7)

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);
}
```

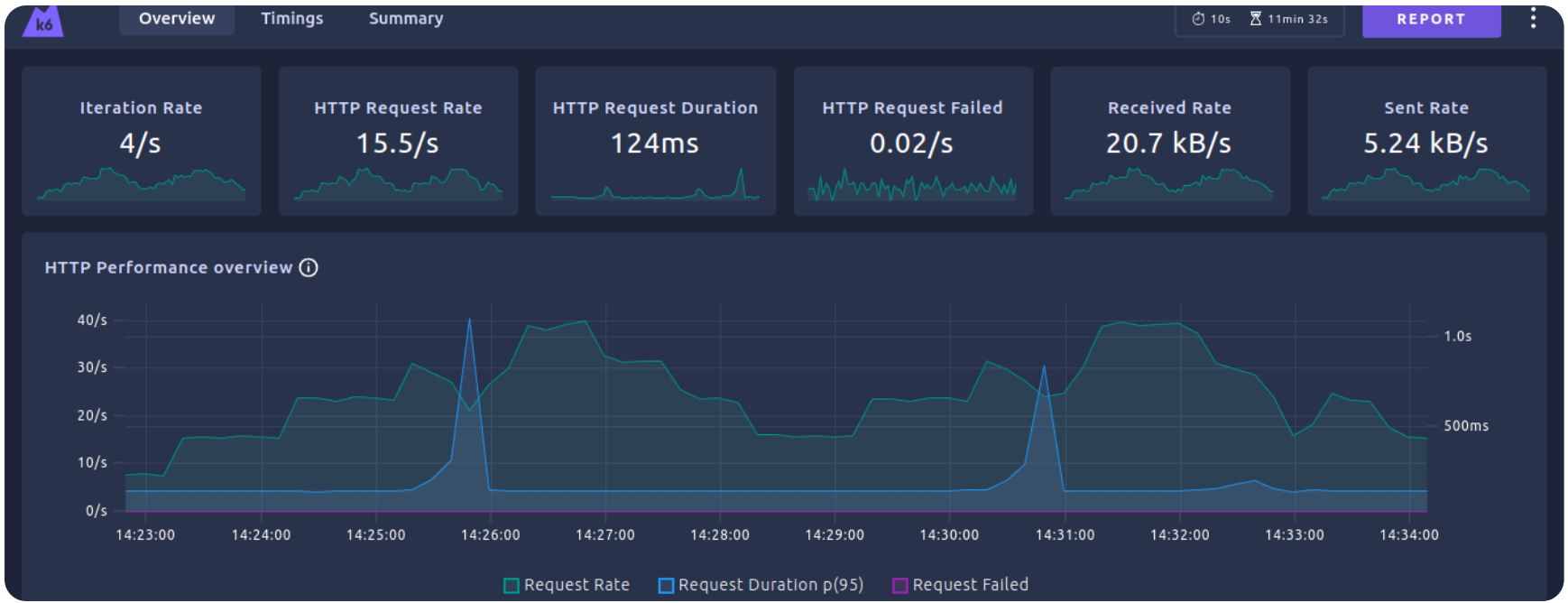


k6: a load generator whose JavaScript script defines what each virtual user does

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

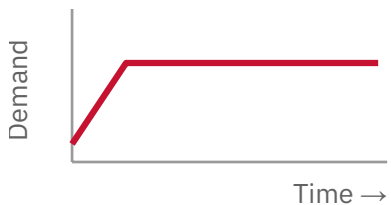
Grafana k6

k6 shows demand and response time together

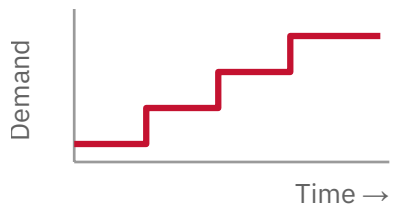


Grafana xk6-dashboard · original example run, overview detail

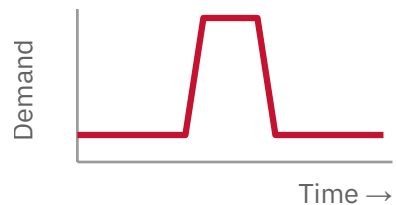
Heavy load can cause a failure without any component breaking



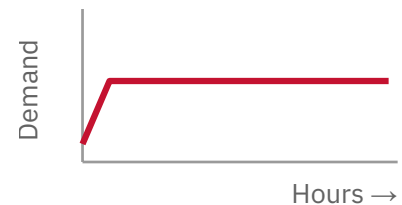
Load test
Can it handle expected peak demand?



Stress test
At what demand does it break, and how?



Spike test
Can it absorb a sudden surge?



Soak test
Does anything leak over hours?

Inject the fault **under realistic load**: an idle system can hide failures that appear when it is busy

Limit the blast radius so that a wrong hypothesis harms few users

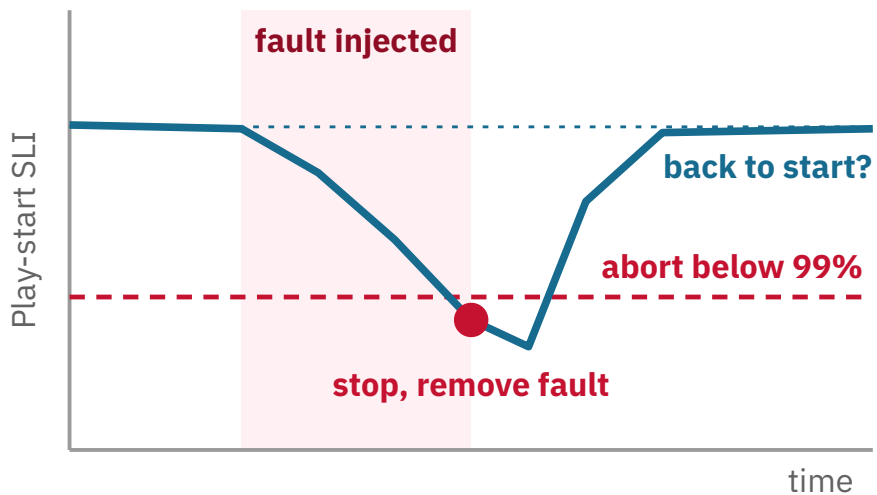
Blast radius: the set of users and systems an experiment can affect



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

Netflix's ChAP limits each group to about 1% of members and all experiments to 5% of a region's traffic, on weekday business hours, never during a failover.

An abort condition ends the experiment before it costs more than agreed



Abort condition: an observable threshold, agreed before the experiment starts, that ends it at once

- Base it on the SLI or error budget
- **Name** who stops the experiment
- **Remove** the fault quickly
- **Check** it is back where it started

Hardening the System

What do we change when an experiment finds a weakness?

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 would you stop the failures you ranked last lecture from reaching users?





KEY CONCEPT

Resilience pattern

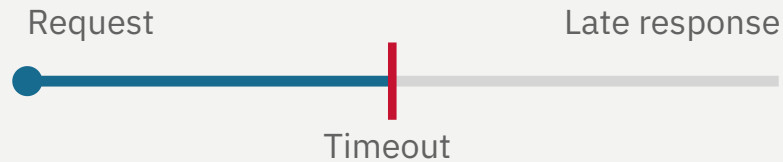
A design element that contains a failure so it does not spread to the rest of the system or to the user.

"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

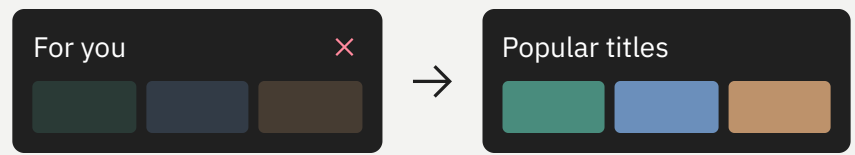
Fail fast

Limit the wait for a slow dependency.



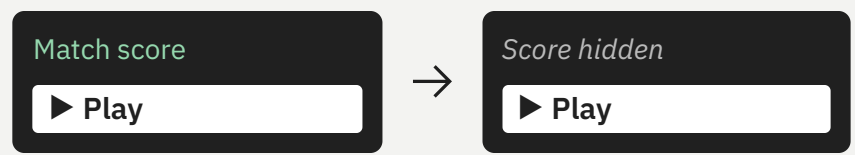
Fallbacks

Use cached or non-personalized results.



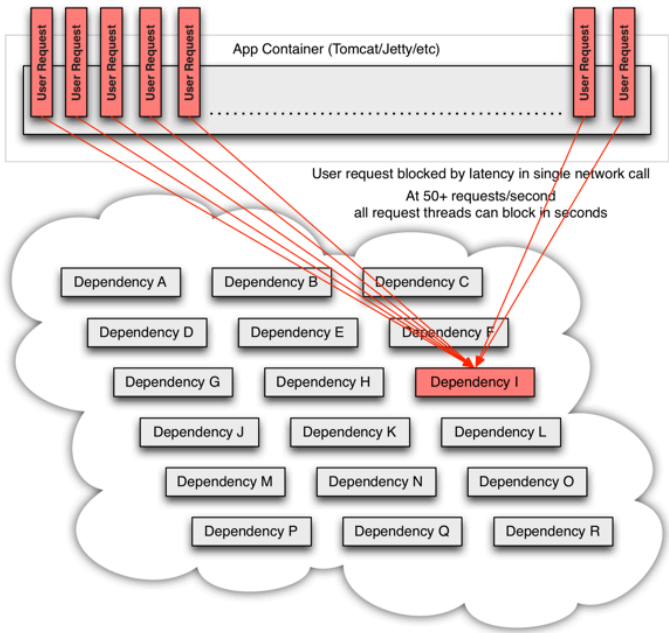
Feature removal

Hide an optional feature when it is slow.



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

One slow dependency can hold every request thread

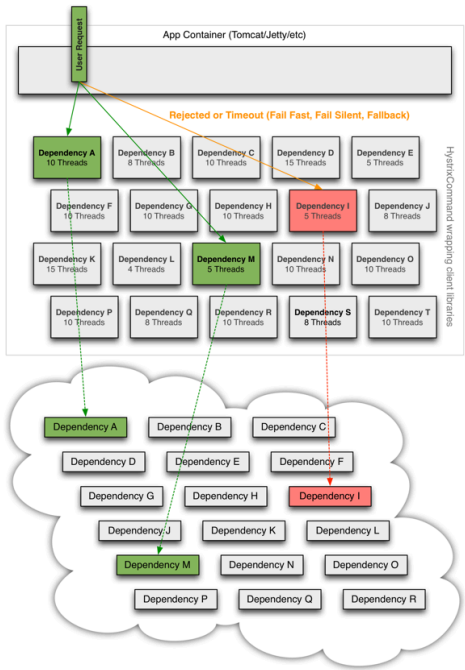


Netflix Hystrix wiki (Apache 2.0)

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

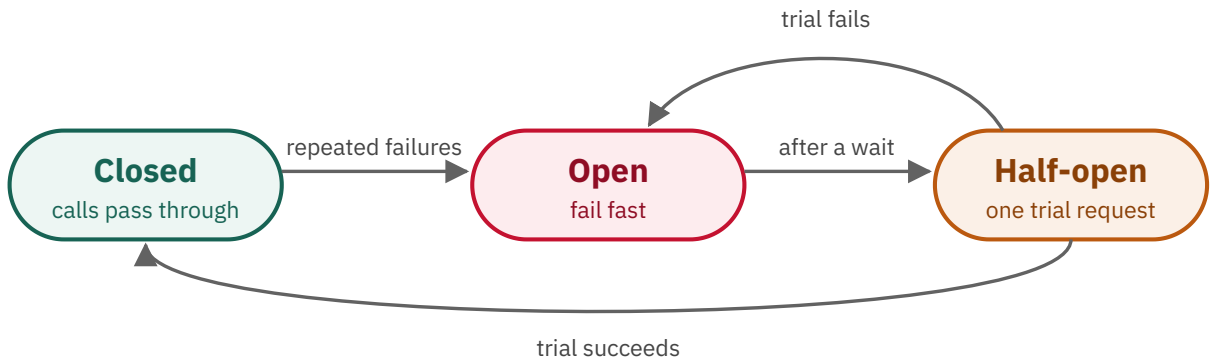
Bulkheads give each dependency its own threads



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



- Repeated failures open the breaker.
- After a wait, allow a trial request.
- If bookmarks is unavailable, start playback from the beginning.

Graceful degradation keeps the core journey working

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.

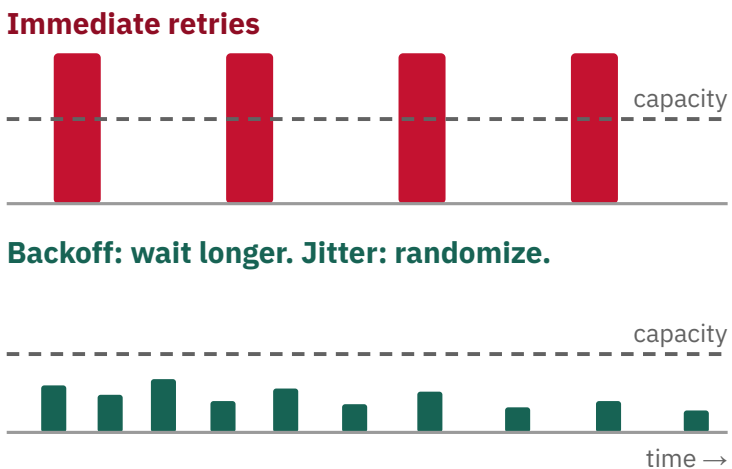
Prefetching
Drop before playback.

Logging
Drop first in this example.

Netflix · Prioritized load shedding

Retries recover from brief faults but multiply load during long ones

⚠ Retries arrive in bursts



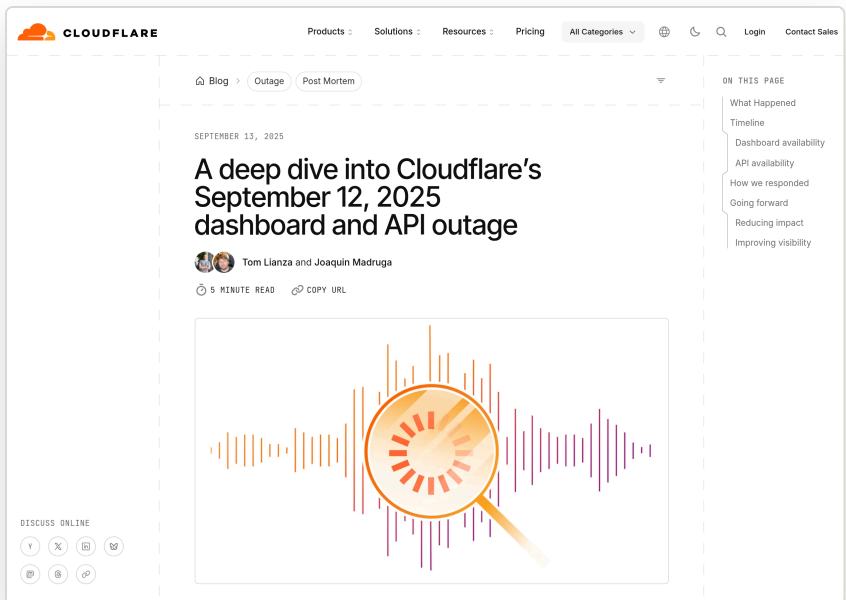
⚠ Retries multiply across layers

Device	●●●	3 attempts
API	●●●●●●●●●	× 3 = 9
Mid-tier	●●●●●●●●●●●●●●●●●●●●●●●●●●●	× 3 = 27

27 requests to bookmarks from one tap

Cap attempts and retry at **one layer only**.

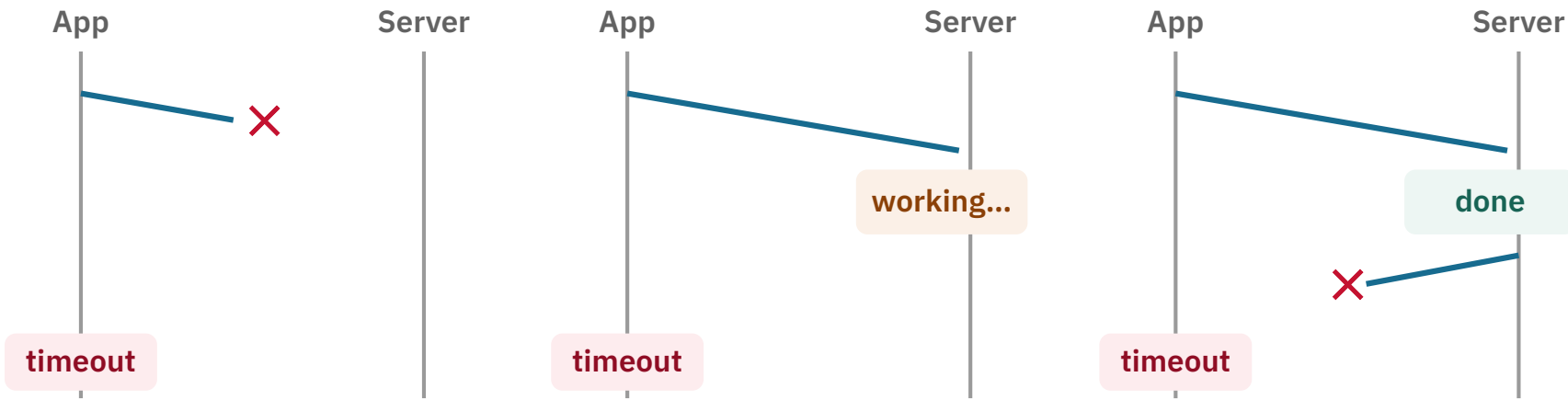
⚠ Unbounded re-requests can overwhelm a shared dependency



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 client whether the request happened



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



KEY CONCEPT

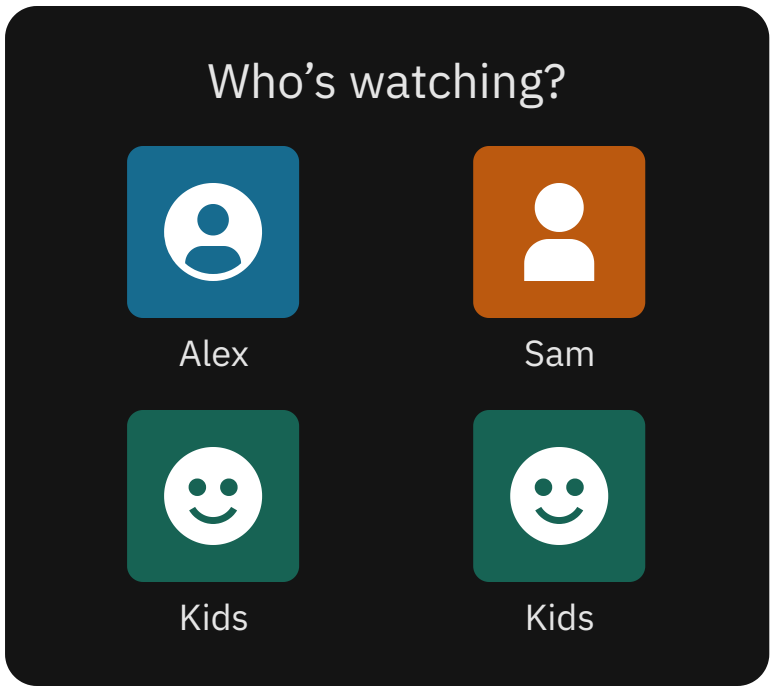
Idempotent operation

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**

Recognize retried writes with an idempotency key

Idempotency key: the client generates one unique key for each intended operation and reuses it on every attempt.

The client sends the key

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

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

- **Client:** generate once; reuse on retries.
- **Server:** recognize the key; return the saved result.

The server checks the key first

Attempt 1 — **New key.** Create the profile and record the key with its response.

`4f9b2c1e-77d0` → 201 Created · “Kids”

Retry — **Key seen.** Return the recorded response.

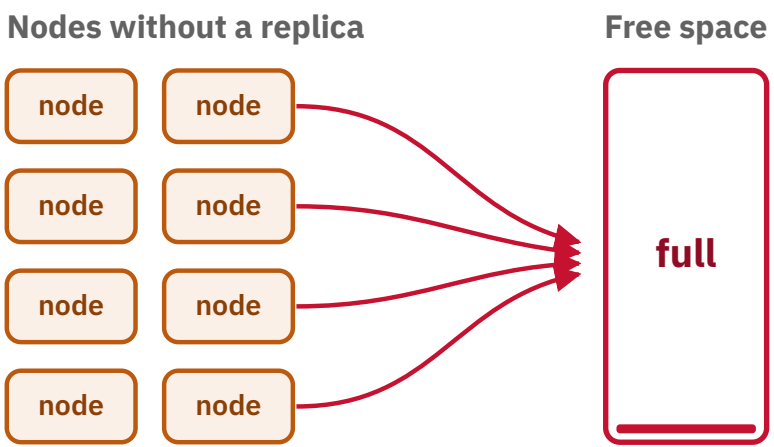
Stripe · Designing robust and predictable APIs with idempotency

Choose the pattern from the failure it must contain

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, because a timeout alone turns slowness into errors unless something replaces the result.

⚠ A defense can make an outage worse



- In April 2011, a network mistake cut many AWS EBS storage nodes off from their replicas
- Each node **copied its data to a new replica**, as designed
- Together they **filled the cluster** and kept retrying, leaving 13% of volumes stuck
- **Lesson:** After adding a defense, rerun and look for **new effects**

AWS · Summary of the Amazon EC2 and RDS Service Disruption (April 2011)

Automate experiments so they keep running 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

Generated experiments, by priority

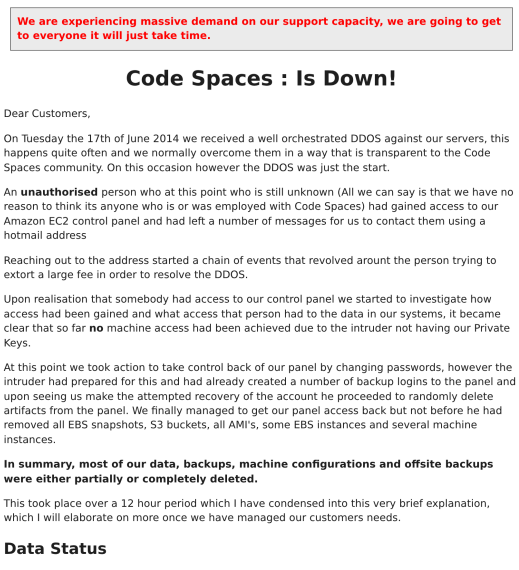
1	**Bookmarks** fails	Held
2	**Ratings** times out	Refuted
3	**Gallery** fails	Running
4	**Personalization** slow	Queued

Queue starts again after each deploy

Disaster Planning

What do we prepare for failures our defenses cannot absorb?

2014: Code Spaces shut down after an attacker deleted its data and its backups



We are experiencing massive demand on our support capacity, we are going to get to everyone it will just take time.

Code Spaces : Is Down!

Dear Customers,

On Tuesday the 17th of June 2014 we received a well orchestrated DDOS against our servers, this happens quite often and we normally overcome them in a way that is transparent to the Code Spaces community. On this occasion however the DDOS was just the start.

An **unauthorised** person who at this point who is still unknown (All we can say is that we have no reason to think its anyone who is or was employed with Code Spaces) had gained access to our Amazon EC2 control panel and had left a number of messages for us to contact them using a hotmail address

Reaching out to the address started a chain of events that revolved arount the person trying to extort a large fee in order to resolve the DDOS.

Upon realisation that somebody had access to our control panel we started to investigate how access had been gained and what access that person had to the data in our systems, it became clear that so far **no** machine access had been achieved due to the intruder not having our Private Keys.

At this point we took action to take control back of our panel by changing passwords, however the intruder had prepared for this and had already created a number of backup logins to the panel and upon seeing us make the attempted recovery of the account he proceeded to randomly delete artifacts from the panel. We finally managed to get our panel access back but not before he had removed all EBS snapshots, S3 buckets, all AMI's, some EBS instances and several machine instances.

In summary, most of our data, backups, machine configurations and offsite backups were either partially or completely deleted.

This took place over a 12 hour period which I have condensed into this very brief explanation, which I will elaborate on more once we have managed our customers needs.

Data Status

All svn repositories that had the following url structure have been deleted from our live EBS's

Code Spaces: notice to customers, June 2014

- An attacker took over the company's **AWS account**
- They deleted servers, **backups**, and **offsite backups**

"Code Spaces will not be able to operate beyond this point."



Which disasters should we prepare for first?

KEY CONCEPT

Disaster

A failure the system cannot ride through, so the team must recover rather than wait for its defenses to work.

For a software team: losing the database or its data, a bad change that corrupts data and replicates it, losing the hosting account or region, or losing access to credentials and tools.

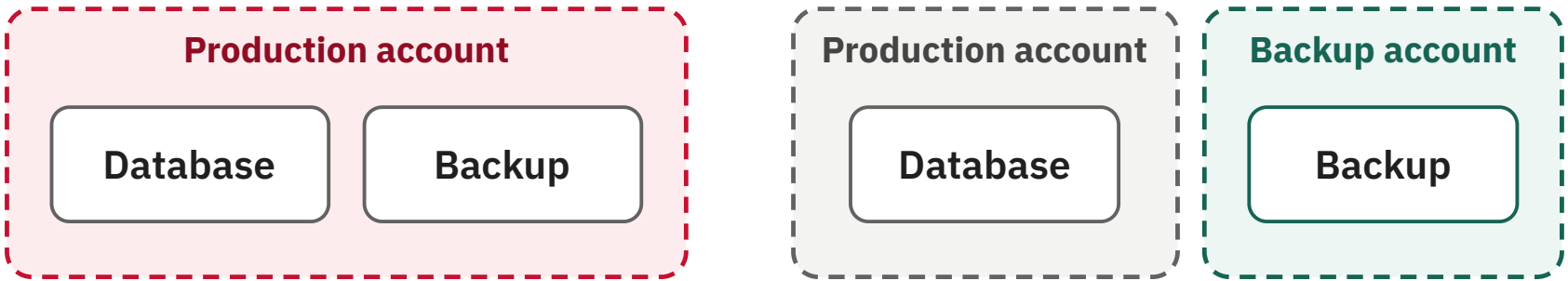
Building Secure and Reliable Systems · Ch. 16, Disaster Planning

Prioritize disaster risks by how likely they are and how much harm they would do

DISASTER RISK	LIKELIHOOD	IMPACT
An engineer deletes production data by mistake	Medium	Critical
A bad migration corrupts data, and replicas copy it	Medium	High
The cloud region hosting the service goes down	Low	High
An attacker takes over the cloud account	Low	Critical

Plan for the highest first, as with your failure-mode ranking on November 4

⚠ Components that share a failure domain fail together

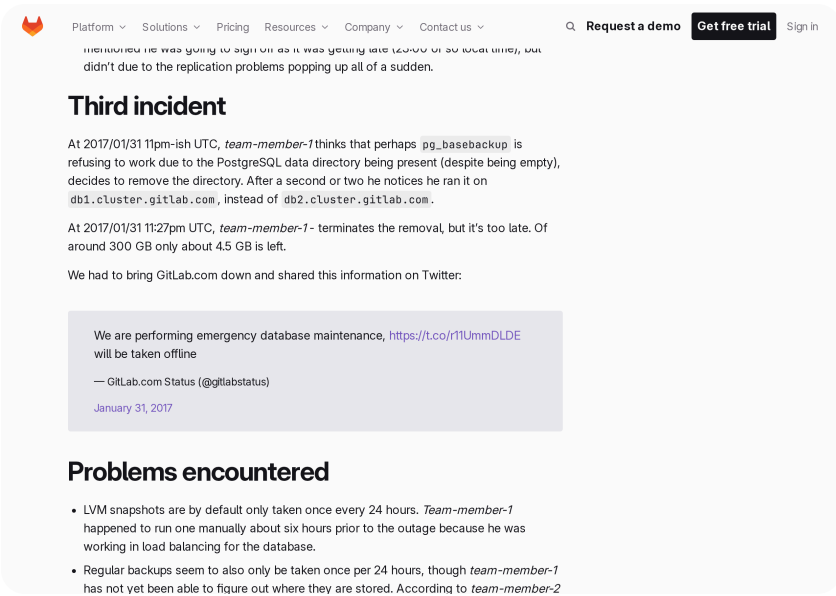


Failure domain: a set of components that a single fault can take down together, such as one disk, one host, one account, or one credential



How much data and time can we afford to lose?

2017: GitLab.com lost six hours of data after an engineer deleted its primary database



Platform Solutions Pricing Resources Company Contact us Request a demo Get free trial Sign in

didn't due to the replication problems popping up all of a sudden.

Third incident

At 2017/01/31 11pm-ish UTC, *team-member-1* thinks that perhaps `pg_basebackup` is refusing to work due to the PostgreSQL data directory being present (despite being empty), decides to remove the directory. After a second or two he notices he ran it on `db1.cluster.gitlab.com`, instead of `db2.cluster.gitlab.com`.

At 2017/01/31 11:27pm UTC, *team-member-1* - terminates the removal, but it's too late. Of around 300 GB only about 4.5 GB is left.

We had to bring GitLab.com down and shared this information on Twitter:

> We are performing emergency database maintenance, https://t.co/r11UmmDLDE will be taken offline
>
> — GitLab.com Status (@gitlabstatus)
>
> January 31, 2017

Problems encountered

- LVM snapshots are by default only taken once every 24 hours. *Team-member-1* happened to run one manually about six hours prior to the outage because he was working in load balancing for the database.
- Regular backups seem to also only be taken once per 24 hours, though *team-member-1* has not yet been able to figure out where they are stored. According to *team-member-2*

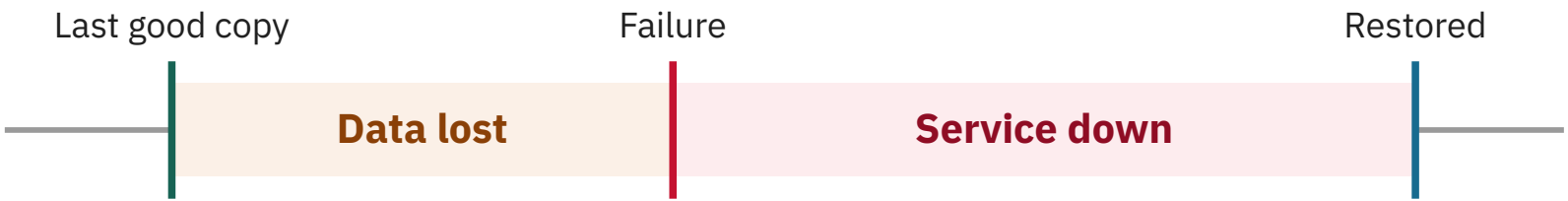
GitLab · GitLab.com database incident, February 1, 2017

- Replication to the **secondary** database broke late at night
- Fixing it, an engineer wiped the **primary** by mistake
- GitLab.com went **offline** to restore it

"Of around 300 GB only about 4.5 GB is left."

GitLab · incident report, February 1, 2017

Each service sets how much data loss (RPO) and downtime (RTO) it can accept



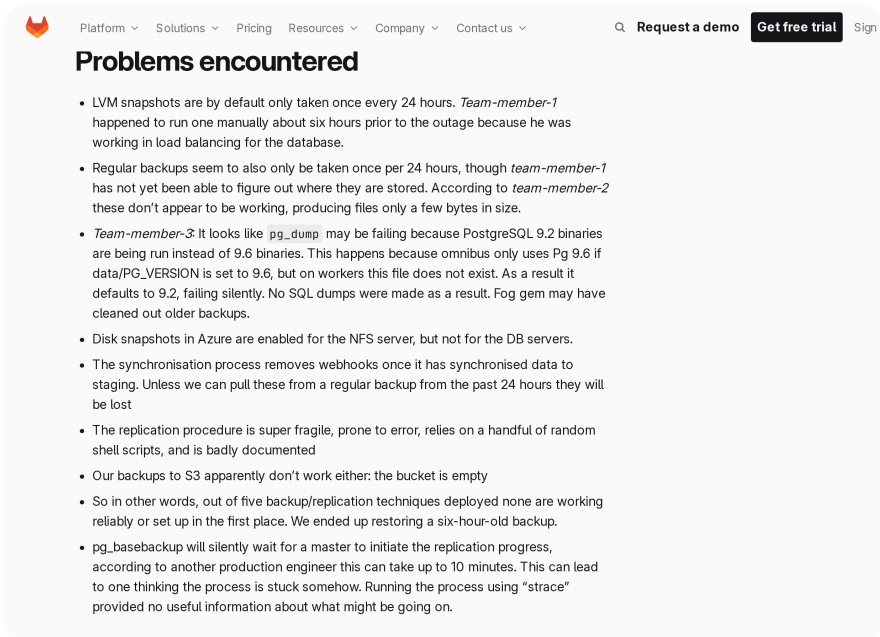
RPO bounds this
GitLab 2017: about 6 hours

RTO bounds this
GitLab 2017: about 18 hours

Recovery point objective (RPO): the most data, measured in time, the service can afford to lose. How often you back up, or how continuously you copy changes, bounds it.

Recovery time objective (RTO): the longest the service can afford to be down. How fast you restore bounds it.

Teams find out whether a backup works by restoring it



Platform Solutions Pricing Resources Company Contact us Request a demo Get free trial Sign i

Problems encountered

- LVM snapshots are by default only taken once every 24 hours. *Team-member-1* happened to run one manually about six hours prior to the outage because he was working in load balancing for the database.
- Regular backups seem to also only be taken once per 24 hours, though *team-member-1* has not yet been able to figure out where they are stored. According to *team-member-2* these don't appear to be working, producing files only a few bytes in size.
- *Team-member-3*: It looks like `pg_dump` may be failing because PostgreSQL 9.2 binaries are being run instead of 9.6 binaries. This happens because omnibus only uses Pg 9.6 if data/PG_VERSION is set to 9.6, but on workers this file does not exist. As a result it defaults to 9.2, failing silently. No SQL dumps were made as a result. Fog gem may have cleaned out older backups.
- Disk snapshots in Azure are enabled for the NFS server, but not for the DB servers.
- The synchronisation process removes webhooks once it has synchronised data to staging. Unless we can pull these from a regular backup from the past 24 hours they will be lost
- The replication procedure is super fragile, prone to error, relies on a handful of random shell scripts, and is badly documented
- Our backups to S3 apparently don't work either: the bucket is empty
- So in other words, out of five backup/replication techniques deployed none are working reliably or set up in the first place. We ended up restoring a six-hour-old backup.
- pg_basebackup will silently wait for a master to initiate the replication progress, according to another production engineer this can take up to 10 minutes. This can lead to one thinking the process is stuck somehow. Running the process using "strace" provided no useful information about what might be going on.

GitLab · GitLab.com database incident, "Problems encountered"

- **Daily database dumps failed:** the backup tool used PostgreSQL 9.2 against a 9.6 database.
- **Failure emails were rejected:** email authentication was misconfigured, so the team never received the warnings.
- **Other recovery options were missing:** database disk snapshots were off, and the S3 backup bucket was empty.
- **The lucky break:** an engineer had manually taken a snapshot about six hours earlier to refresh staging for testing. GitLab restored from that copy.

Keep backups out of production's reach, and restore them on a schedule

Retention: how long each backup is kept before it is deleted

✗ Instead of

Nightly backups in the production account, kept for 7 days, never restored, with failure emails nobody reads

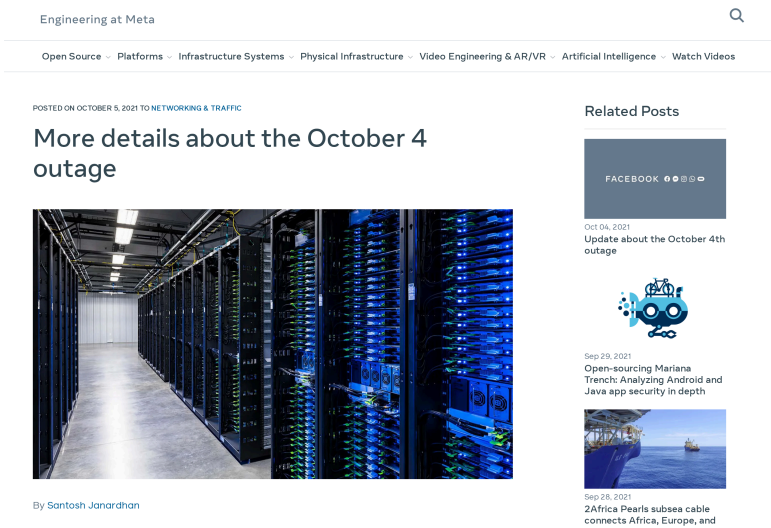
✓ Do this

Back up to an account production cannot delete from, keep each copy longer than bad data could go unnoticed, and restore one automatically every day, paging someone if it fails



Can responders still reach the system when it fails?

Responders need access and tools that do not depend on the failed system



Engineering at Meta · More details about the October 4 outage

- A configuration change cut Meta's data centers off from the internet
- The internal tools engineers normally use went **down too**
- Engineers had to go **on site**, and physical security slowed them down

Prestage access and tools outside the failure domain they must recover

- **Prestaging:** arranging credentials, tools, and contacts before a disaster
- **Break-glass credential:** a sealed emergency login whose use raises an alert

✗ Instead of

The restore runbook, credentials, and on-call list live in a wiki in the database's cloud account

✓ Do this

Keep the runbook, a break-glass credential, and the contacts outside that account, and test routinely



How does the person on call know what to do?



KEY CONCEPT

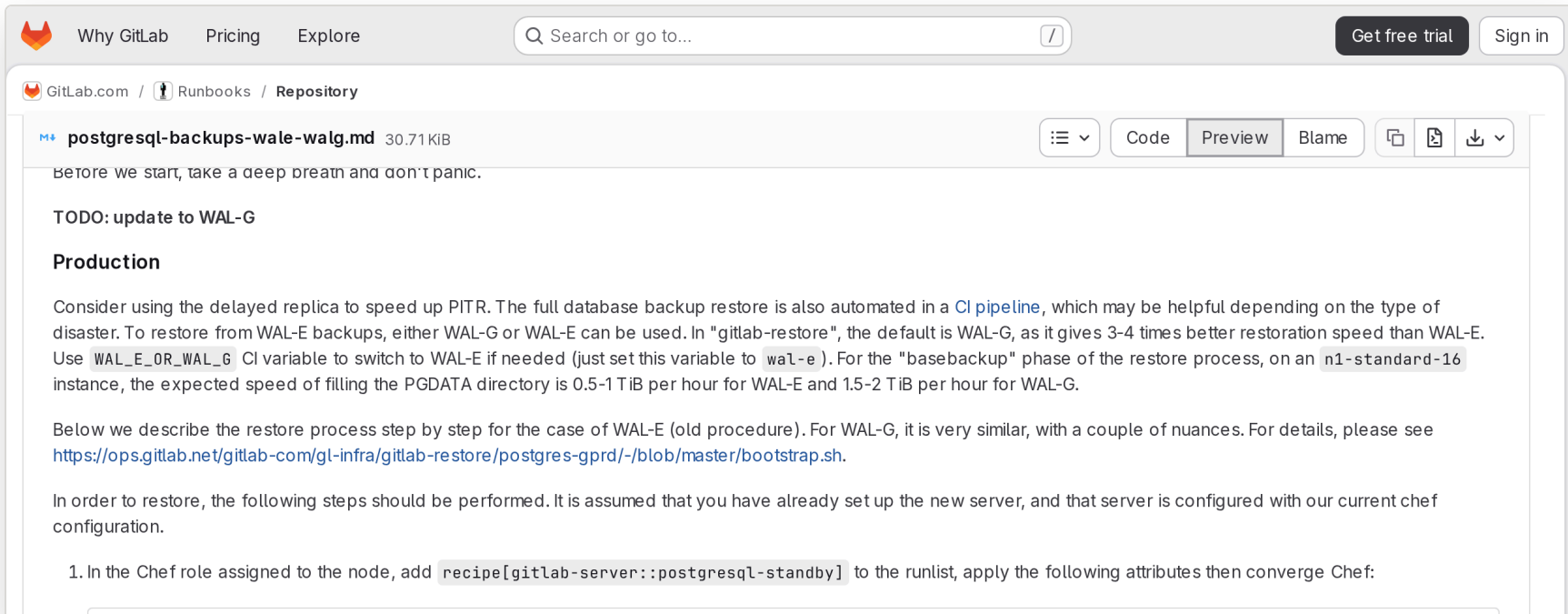
Runbook

A written procedure for a known failure: how to diagnose it, the steps to mitigate or recover, how to check that each step worked, and when to escalate.

Google found that recording best practices ahead of time gives **roughly 3× faster recovery** than improvising.

Google SRE book · Introduction

GitLab publishes its database restore runbook



GitLab runbooks · PostgreSQL Backups: WAL-G, restore section

Write each runbook step with a condition, an expected result, and an escalation

GitLab's restore runbook

- ✅ States the expected restore speed
- ✅ Restores are tested twice a day
- ❌ Steps still marked `TODO`
- ❌ No rule for the delayed replica
- ❌ No one to escalate to

Write each step as

> **If** the primary's data is lost,
>
> **do** restore into a new, isolated database,
>
> **expect** sampled issues to match the backup within 10 hours;
>
> **otherwise escalate to** the database on-call



How do we know the plan will work?

A disaster plan is trustworthy only after it has been rehearsed

Tabletop exercise

The team walks through a scenario on paper to find **gaps in the plan**

"The database is gone. Who restores it, from which backup, with whose credentials?"

Recovery drill

The team really restores a backup or fails over, to find **gaps in the tools and people**

Restore last night's backup into a new database, and time it

Each rehearsal is itself an experiment, so it needs a blast radius, an abort condition, and a way back

Google rehearses disaster recovery every year in its DiRT program

DiRT Exercise Testing Emergency Access

Sometimes we can test the robustness of both reliability and security operations simultaneously. During one of our annual Disaster Recovery Training (DiRT) exercises,[6] SREs tested the procedure and functionality of breakglass credentials:[7] could they gain emergency access to the corporate and production networks when standard ACL services were down? To add a security testing layer, the DiRT team also looped in the signals detection team. When SREs engaged the breakglass procedure, the detection team was able to confirm that the correct alert fired and that the access request was legitimate.

Industry-Wide Vulnerabilities

In 2018, Google received early notice of two vulnerabilities in the Linux kernel, which underpins much of our production infrastructure. By sending specially crafted IP fragments and TCP segments, either SegmentSmack (CVE-2018-5390) or FragmentSmack (CVE-2018-5391) could cause a server to perform expensive operations. By using large amounts of both CPU and wall-clock time, this vulnerability could allow an attacker to obtain a significant scale boost beyond a normal denial-

Building Secure and Reliable Systems · Ch. 16

- **Disaster Recovery Testing (DiRT):** yearly tabletops and real tests, up to taking data centers offline
- One test checked the **break-glass credentials**

> *"could they gain emergency access to the corporate and production networks when standard ACL services were down?"*

Prepare for a disaster before it happens



Rank the risks by likelihood and impact, and look for shared failure domains



Set an RPO and RTO, and keep backups out of production's reach



Prestage access and tools outside the failure domain



Write runbooks with conditions, expected results, and escalations



Rehearse with tabletop exercises and recovery drills

Defend or Recover?

What can keep working during a failure? What must the team recover?

Activity: Analyze one failure for the photo-sharing app



1 What would users see?



2 Could a resilience pattern contain it? Which one, and what would users see instead?

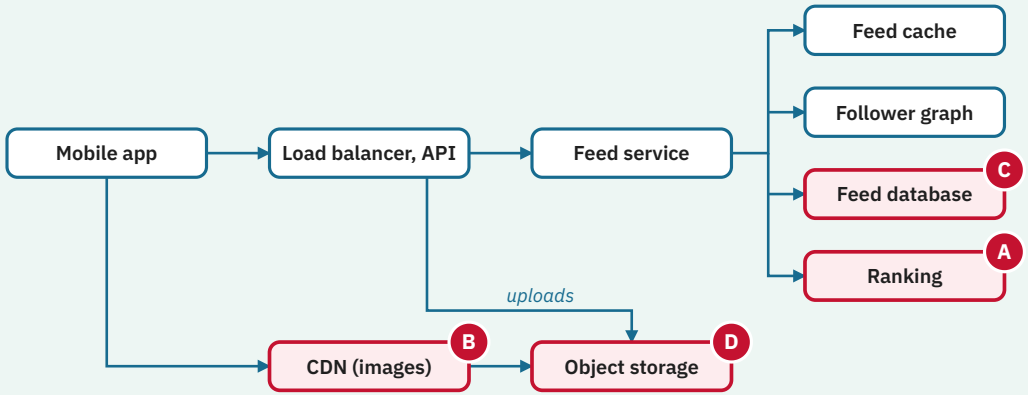


3 What still needs recovery? Propose RPO and RTO targets with assumptions; explain what must be prepared. If no data recovery is needed, explain why.



4 Which rehearsal would show that the defense or plan works? What are its blast radius, abort condition, and way back?

What can keep working, and what needs recovery?



- **A** Ranking slows down
- **B** The CDN stops serving images
- **C** A migration corrupts the feed database
- **D** The storage credential expires

Debrief: What can we contain, and what must we recover?

Contain A

- A resilience pattern
- An experiment that confirms it

B D

May need both

Recover C

- An RPO and an RTO
- A tested restore
- Access prepared in advance

Rehearse both with a blast radius, an abort condition, and a way back

🚀 Today, we learned how to ...

- Design a bounded fault experiment with an abort condition
- Choose a resilience pattern and predict how a rerun should change
- Explain retry storms, and how an idempotency key makes retries safe
- Tell an absorbable failure from a disaster, and rank disaster risks
- Explain RPO and RTO, and why only a tested restore protects data
- Judge whether a runbook is safe to follow