https://www.youtube.com/watch?v=-lHPDx90Ppg

SLO Math

Steve McGhee, Google
San Luis Obispo (SLO), CA

% whoami

- **Google SRE, SRM**: Android, Fiber, YouTube, Cloud
- **Infrastructure Architect** - New Platform, old Product - *Modernize?*
- **Solutions Architect** - DevOps, SRE, Anthos - *Modernize!*
- **Reliability Advocate** - Internal + External

Preamble

What I hope you already know: What an SLO **is**.

What I hope you'll learn: How to **use** SLOs. How **not** to use SLOs.

Heads up: Math Ahead (see title). A bit of probability.

"The Front Door SLO"

Focus on the customer's happiness.

- ➔ Available (enough)
- ➔ Fast (enough)
- ➔ Complete (enough)

Don't think about the serving system (yet).

Meet Expectations
Don't Expect Perfection

https://unsplash.com/photos/QR_vT8_hBZM

Deeper SLOs

"But it's more complicated than that, Steve"

I know.

"My service depends on other teams"

I know.

https://unsplash.com/photos/oEtQ0C6HS2I

~~Bad~~ Naive Math

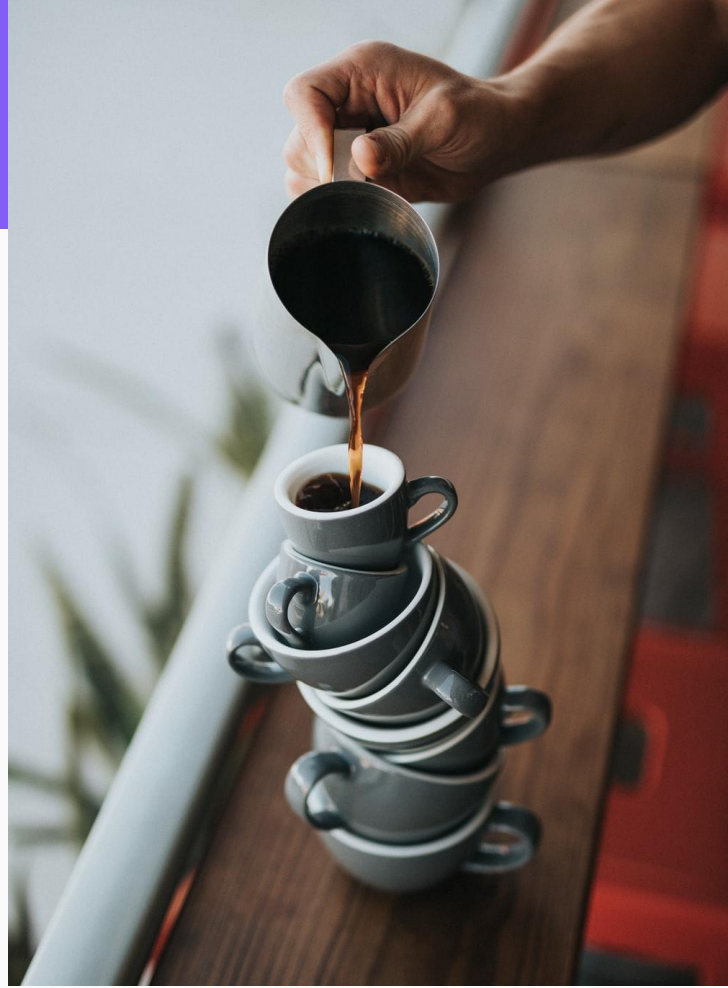
my users expect 99.0%

so my webserver should be 99.**9**%

so my database should be 99.**99**%

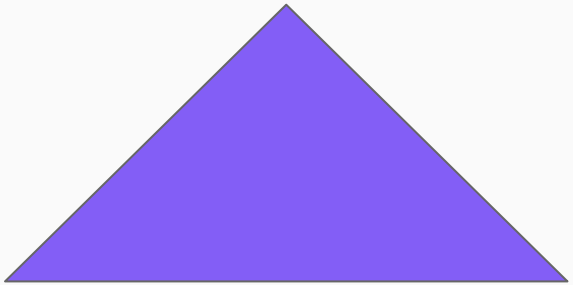
so my infrastructure should be 99.**999**%

... but what if i have *more* layers? 🤯



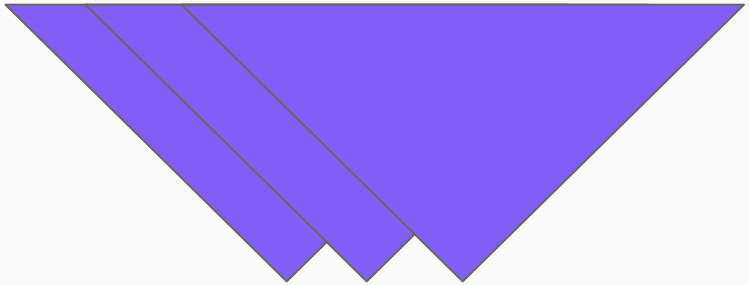
https://unsplash.com/photos/3kZpELkaxHc

Context: The Pyramids



Component-level reliability:

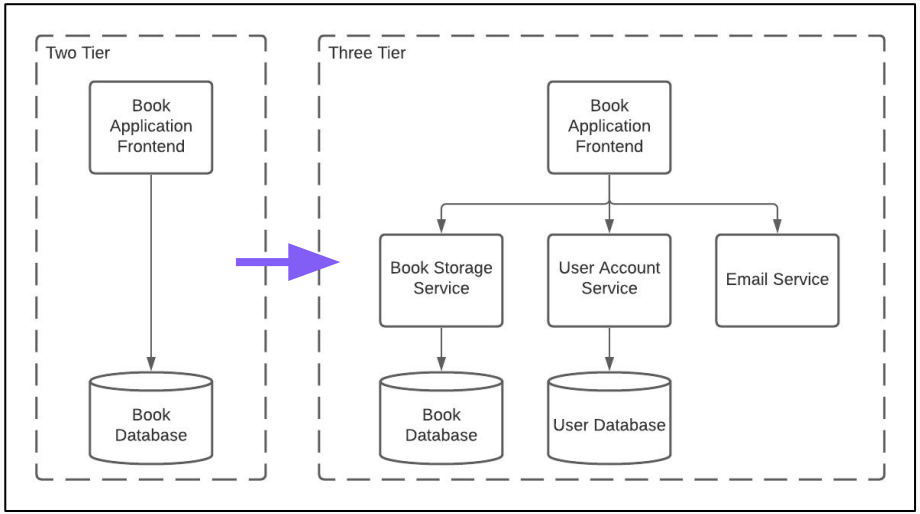
- solid base (big cold building, heavy iron, redundant disks/net/power)
- **each** component up as much as possible
- **total availability** as goal
- "scale up"

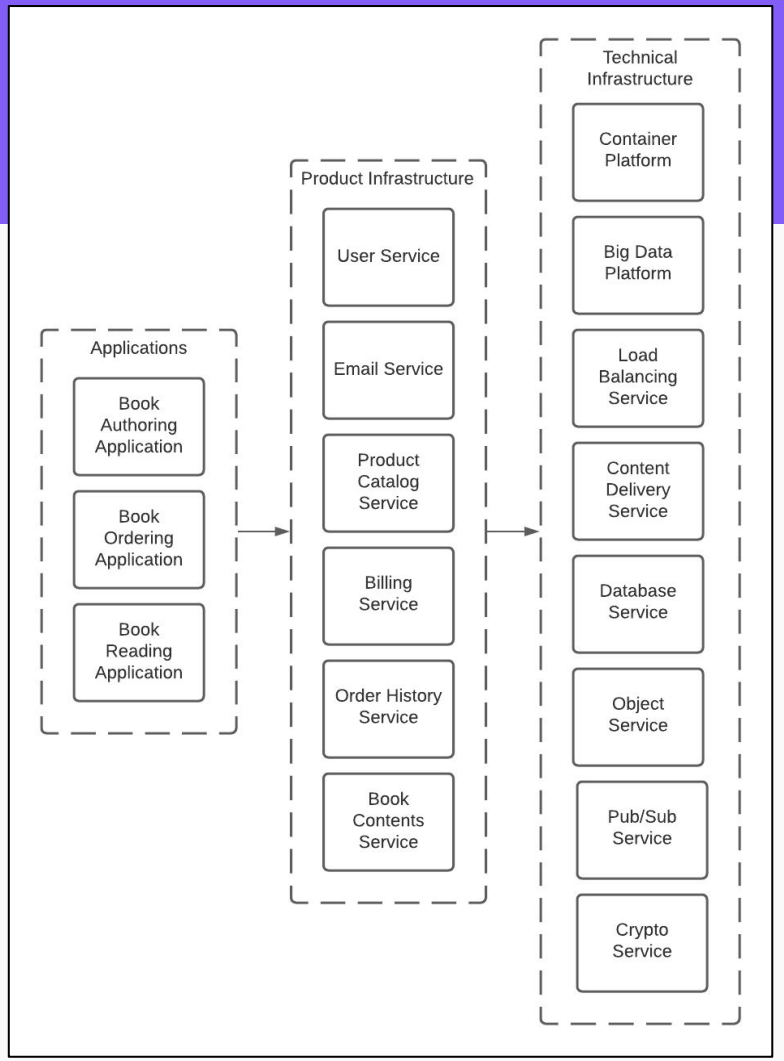


Scalable reliability:

- less-reliable, cost-effective base
- "warehouse scale" (many machines)
- software *improves* availability
- **aggregate availability** as goal
- "scale out"

What Else Has Changed?



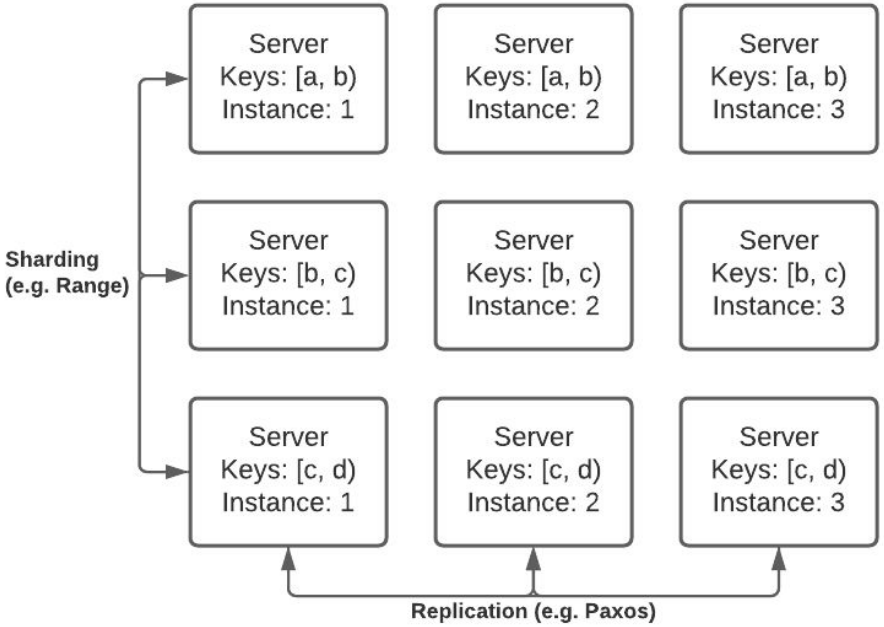


Good Math needs a Model

First we need a baseline model of a Cloud-based **distributed system**.

We need to allow for:

- Scalability (Horizontal, Vertical)
- Sharding, Partitioning
- Replication, Load Balancing



(If this stuff is new to you, don't fret.)

Probability, real quick

Let's call rolling a "1" an outage.

probability of having an outage:
⅙ = 0.1667

probability of NOT having an outage:
⅚ = 0.833 ← "**availability SLO**!"

Now it gets weird:

Four 6-sided dice: `(⅚)^4 = 0.482`

Four N-sided dice: `(N-1)/N) ^ 4`

M arbitrary-sided dice:
$(N_1 - 1/N_1) * (N_2-1/N_2) * (N_M-1/N_M)$

eg: two 6-sided, one 10 sided, one 20 sided dice:
`5/6 * 5/6 * 9/10 * 19/20 = .83 * .83 * .90 * .95 = .59`

https://unsplash.com/photos/QuP5RL_E5oE

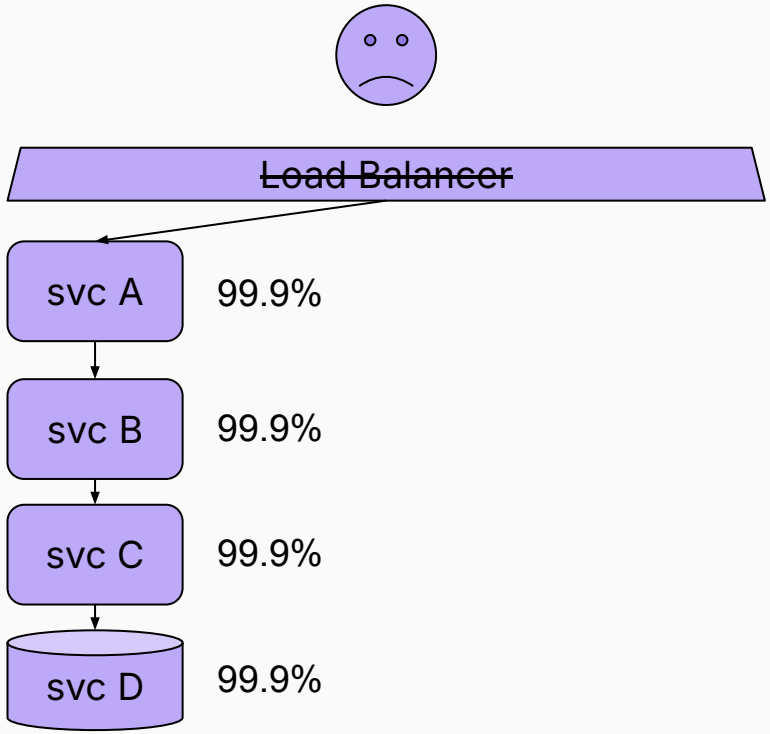
"you'll never do as well as the **worst case single throw**"

Serial Services

What if you have services that depend on each other, in a "straight line"?

3 nines @ depth 4 gets us "2.6" nines:
`(0.999, 0.999, 0.999, 0.999) = 0.999^4 = **99.6%**`

SLO^depth



So what? Your **architecture choices** can have *more* of an impact than the SLOs of your dependencies.

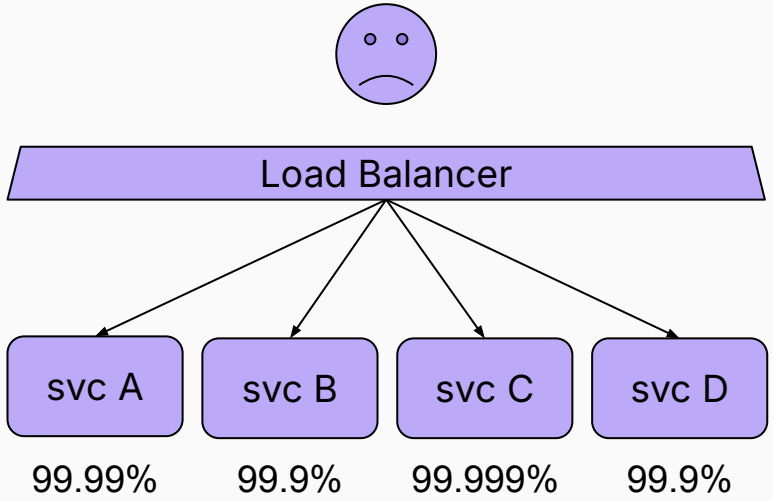
Parallel, Required Services

N services and your app is the **composition of all** of them, they **all still need to be up**!

One failing is **just as bad** as the Serial configuration. oof.

We're still at:

SLO^depth



Or if we have "different-sided dice" it is:
`(0.9999 * 0.999 * 0.99999 * 0.999)` = **`99.789%`**

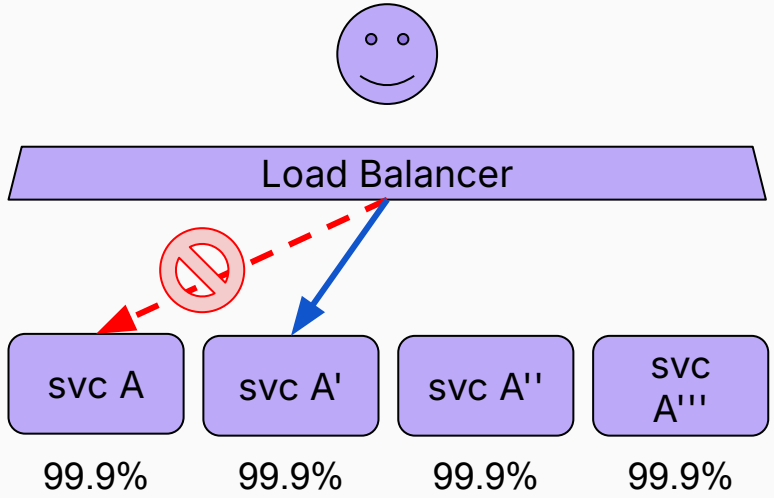
Redundant Services!

What if you have **independent copies of the same service**? As long as **one** is up, you're happy! Now we're talking!

Now your outage only has the probability of all N services failing **at the same time.** (*ahem: presuming automatic retries*)
Our failure_ratio is just: 1 - SLO

Given 4 dice, you have to **roll four ones** in order to fail.
The odds of this are: (1/6 * 1/6 * 1/6 * 1/6) = 0.00077

1 - failure_ratio ^ redundancy




```
1 - (0.1% * 0.1% * 0.1% * 0.1%) =
1 - .001^4 = 99.99999999...%  (12.6 nines!)
```

Set Theory of SLOs

Intersection availability is where **all** dependencies **must** be available.

➔ `SLO ^ depth`

➔ `0.999 ^ 3 = 99.7%`

Union availability is where **at least one** of the dependencies must be available.

➔ `1-(1-SLO) ^ redundancy`

➔ `1-(1-0.999) ^ 3 = 99.9999999%`

Component Availability	Intersection Availability per Number of Components			Union Availability per Number of Components	
	3	10	100	2	3
99%	97%	90%	37%	99.99%	99.9999%
99.9%	99.7%	99%	90%	99.9999%	99.9999999%
99.99%	99.97%	99.9%	99%	99.999999%	99.9999999999%
99.999%	99.997%	99.99%	99.9%	99.99999999%	99.9999999999999%

Bottlenecks

So why aren't we swimming in nines?

- Network, Load Balancing is a nines-lynchpin. Hard to fix. Hard to own.
- Changes, Mistakes, Churn, Shortcuts.
  - Own the end-to-end SDLC, every part matters.
- Shallow Understanding / Striving for Over-Simplified Learnings
  - (the details of an outage matter)
  - (in fact, they're **all that matters**)

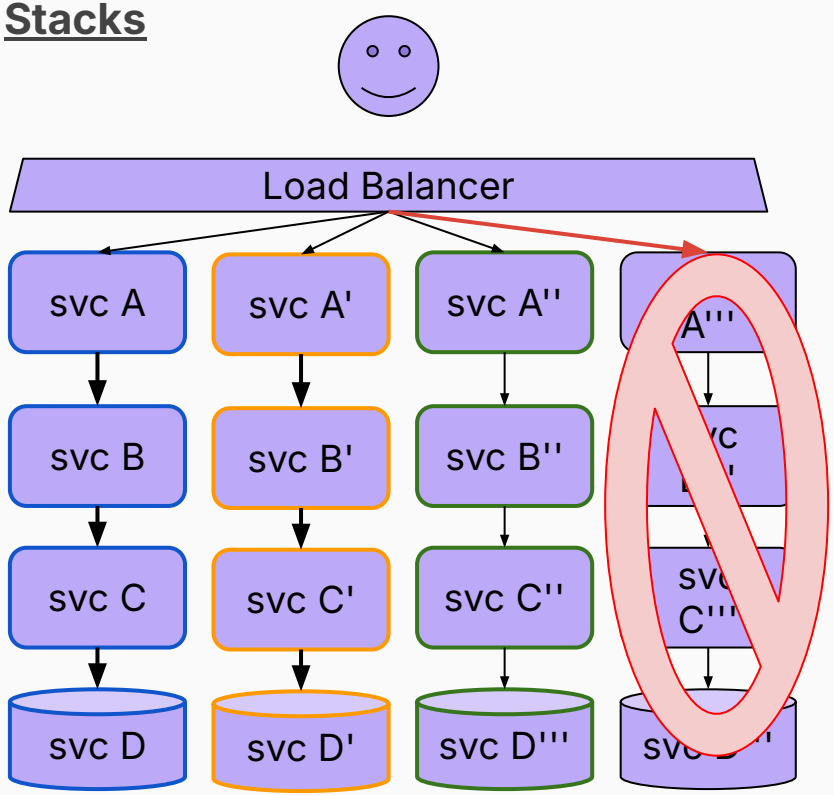
Now What?

1) **don't fret** about the downward implications of your SLO choices.
   → consider your customer's happiness first and foremost
   a) set **customer appropriate goals** and your stack will work to meet them
   b) use outages to understand the details of resilience engineering needed

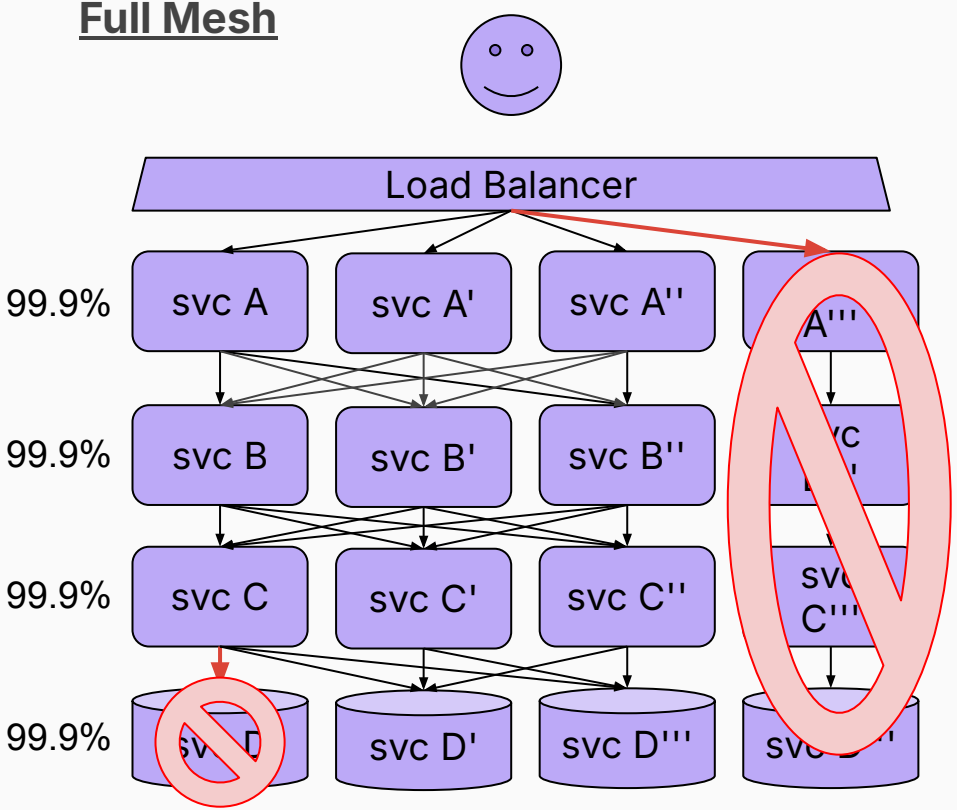
2) help infrastructure teams understand the new world:
   resilient software can make infra easier
   a) it's a team effort, own the whole problem

Two Usable Models

Stacks



Full Mesh



Gnarly Details

Not recommended: YOLO, "megalith"
Maximum Resilience: full-mesh.

Costs:

- more computers, storage!
- operational complexity! (stacks simpler)
- consistency, sharding, replication issues

Further Reading:

- Failure domains (physical, logical)
- Regions/Zones default FDs
- Failure Modes
- Graceful degradation

https://unsplash.com/photos/eCfCIwtKdsg

Wait, that's Way Too Simple / False!

Correct! Some **Fallacies** (so far):

1) SLOs must get tighter with depth

Solutions:

1) Resilience via Engineering! You can build **more reliable things** on top of **less reliable things**

2) I actually control the entire stack

2) Do you own the loadbalancer? The mobile tower? The battery?

This Bears Repeating

You can build
more reliable things
on top of
less reliable things

a simple example: RAID. see: *The SRE I Aspire to Be*, @aknin SREconEMEA 2019

Closing

A colleague asked me:

"When should you define an SLO for a system vs it's components?"

I hope what you take away from this talks is:

You should design a **system** at "the front door" but it's a common mistake to follow Conway's Law and define it at team boundaries, then get frustrated by the "bad math" that ensues.

Build a platform that lets you focus on customer happiness. All else will follow.

https://unsplash.com/photos/dBaz0xhCkPY

https://unsplash.com/photos/D-jLxBtEwaA

Error budget burn-down chart

		Percent of Service Unavailable			
		0.1%	**1%**	**10%**	**100%**
Availability SLO	**90%**	Forever	Forever	Forever	9d
	99%	Forever	Forever	9d	21h
	99.9%	Forever	9d	21h	2h
	99.99%	9d	21h	2h	12m
	99.999%	21h	2h	12m	1m

DR Math

"Holdback" = 1/N

As you increase N:

DCs can each run "hotter" now (better utilization)

Better global spread should also result in better latency (faster experience) to global users

If each DC is totally independent, your availability "nines" improve dramatically (and are actually capped by the loadbalancer/network),