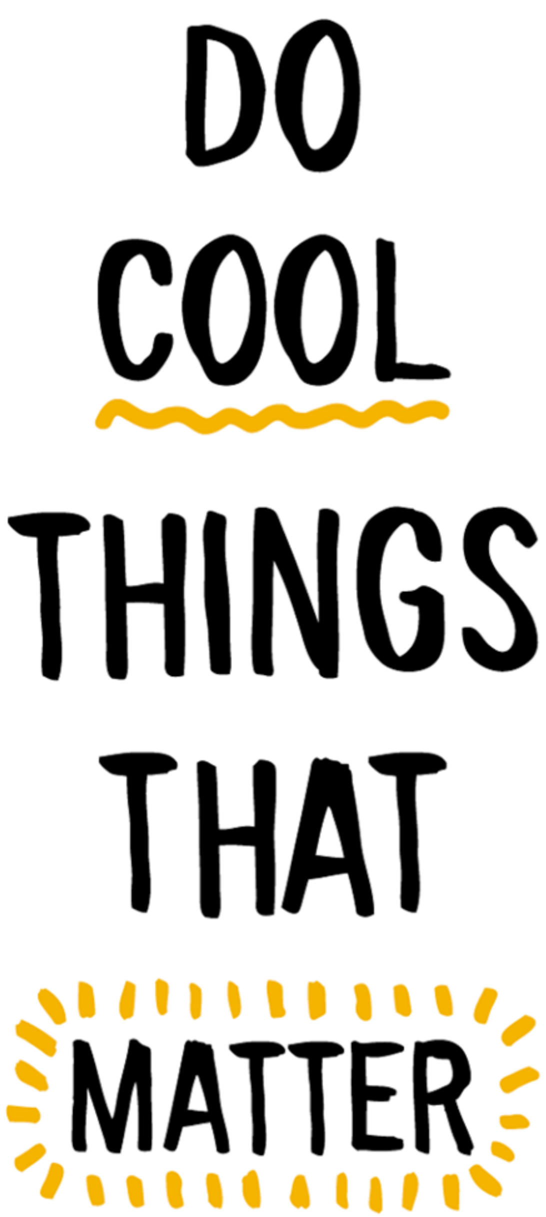


Engineering Reliability:

How to build reliable large-scale services

2014

Seongbae Park
Site Reliability Engineering

Who am I ?

2014 - now	Google, SRE, Identity
2008 - 2014	Google, SRE, Websearch
2006 - 2008	Google, compiler optimization
1999 - 2006	Sun Microsystems, compiler optimization

Technical Lead, Search SRE

DO COOL THINGS THAT MATTER

What is
Site Reliability Engineering?

Site Reliability Engineering (SRE)

- Separate organization from the software engineering organization
- Tasked with maintaining Google's reliability
- Users expect Google to be always up and fast
    - ... and we try to fulfill that expectation
- Maximize the rate at which new features and new products can be delivered to our users
    - without breaking the Google

Serve request

reliably
quickly
cheaply

(Pick all three.)

Organizational Reliability

Organizational Incentive

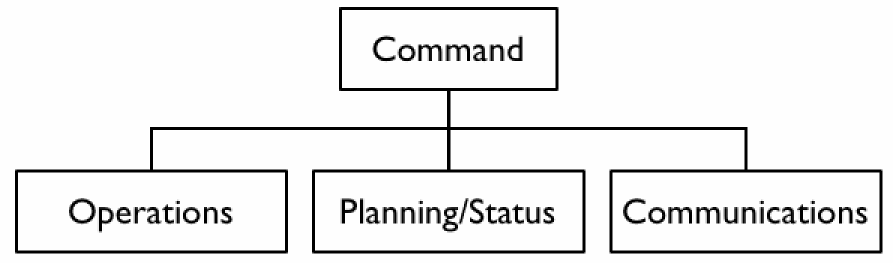
- Humans and organizations optimize for their own incentive.
- No change means best reliability.
- Trade-off between reliability vs speed of change
- Trade-off between cost vs speed of change
- Trade-off between cost vs performance
- Trade-off between engineering complexity vs reliability
- Trade-off between human operator vs automation

Service Level Agreement

- A measurable metric
  - e.g. 99.99% of requests successful within 1 seconds over the quarter.
  - 99.999% of requests successful within 50ms over the quarter.
- Development team and Site Reliability team agrees *beforehand*.
- The rest is considered “failure budget”.
  - 99.99% service, with 100M daily queries, over a quarter
    => 900,000 queries can fail over the quarter
- Measured and tracked all the time.

Incident Managment

- Incident Management Framework
    - Command
    - Operate
    - Communicate
- Temporary
- Incident Commander
    - Operations Lead
        - Typically oncaller who handles the page
    - Communications Lead
    - Planning/Status Lead



Postmortem

- Any events with (potential or real) big impact, or high complexity
    - A summary of the events
    - A full timeline of the problem - all factual information
        - background, contributing factors, how it was discovered, mitigated, fixed
    - User Impact
    - A root cause
    - What worked and what didn’t

The goal of postmortem is to learn and improve, not to assign blame. All human mistakes need to be captured accurately, and management does not blame mistakes. Focuses on how to prevent similar occurrence in all problems in the same **class** of problems.

Training

- Wheel of Misfortune
  - Simulated incident management
  - Actual or imagined incidents
- Annual large-scale disaster test
  - Simulated or actual
- Emergency drills

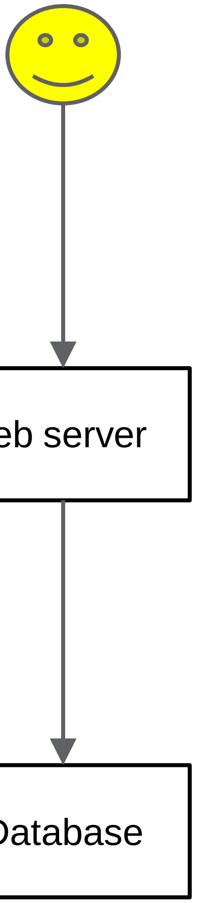
It's All About Scale

Anatomy of large-scale web service

It compiles, so it must work!

- Suppose you just wrote a web service
- It's running on a server with a MySQL db behind it
- You can point your web browser to it
- That means you're done, right?

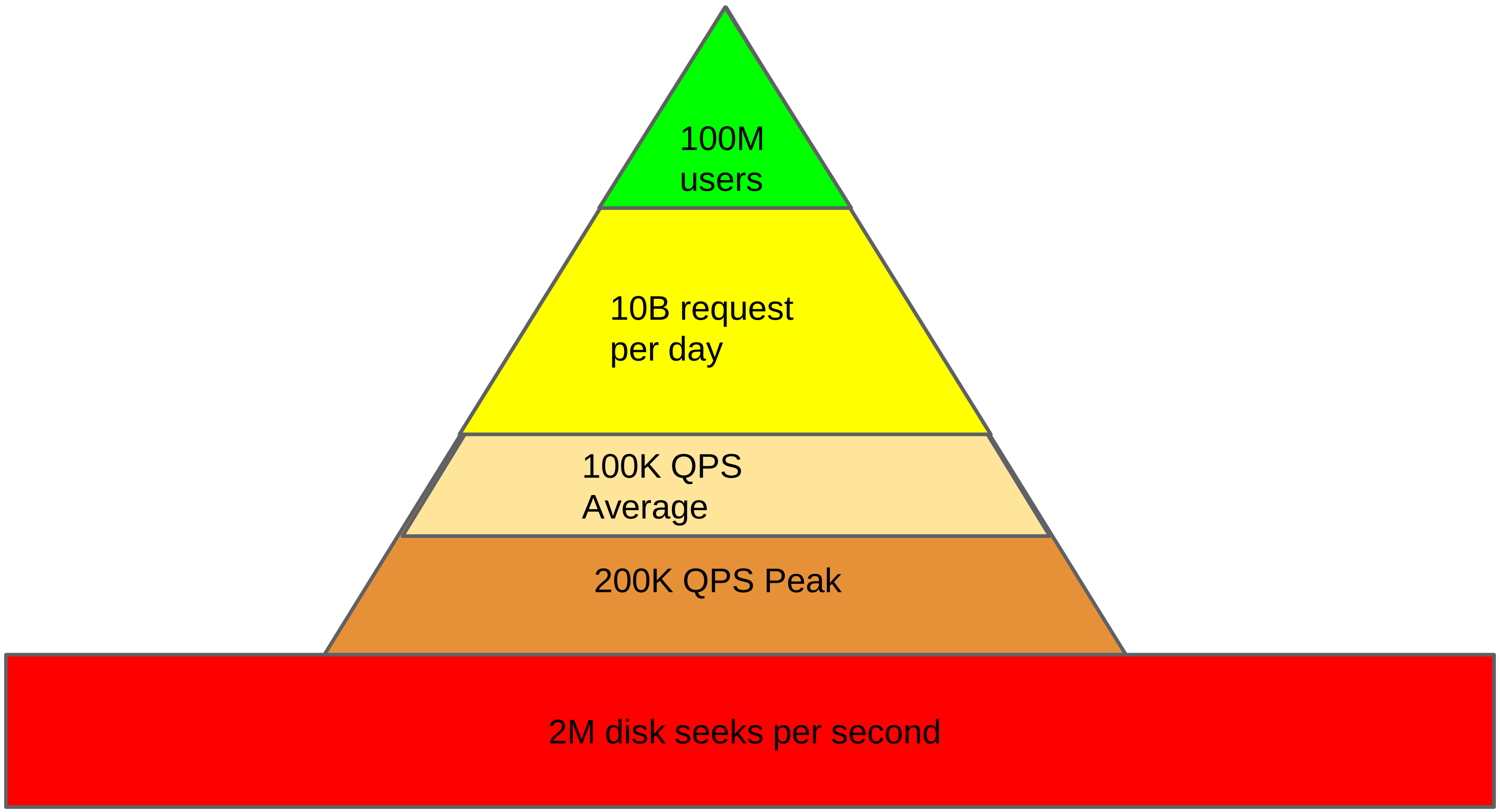
- Right?


Web server
Database

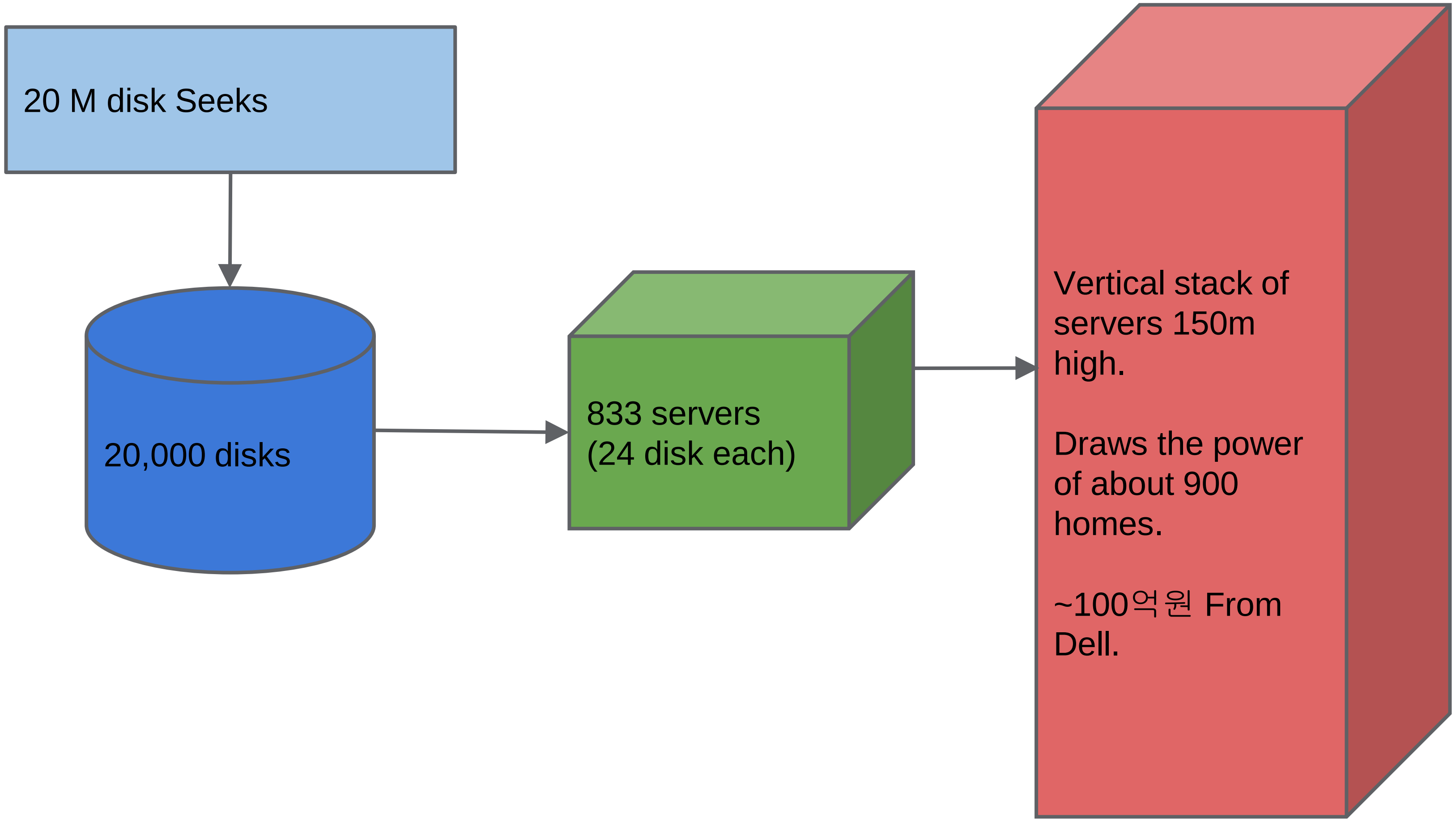
Scale the number of users

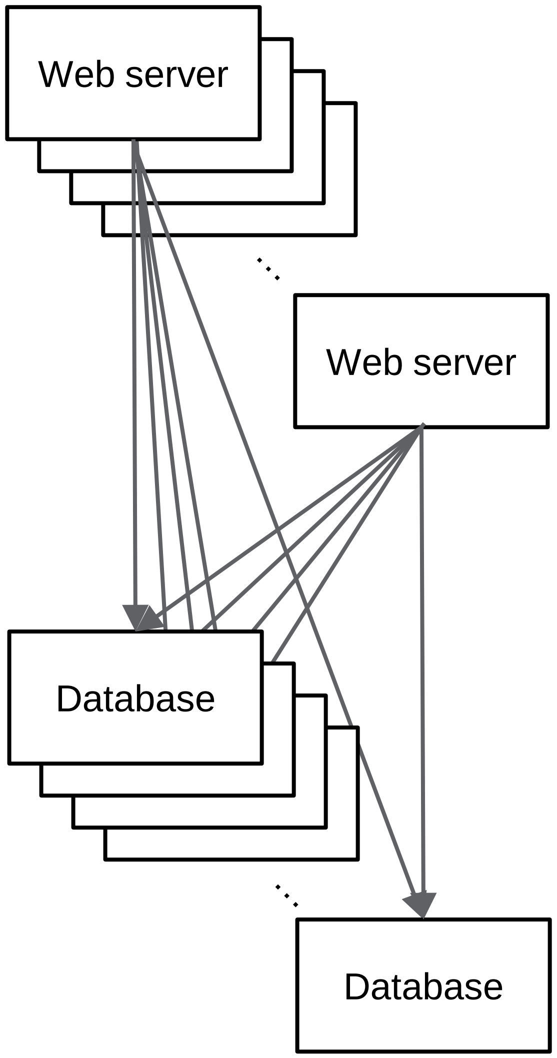
- Now imagine your web application has to serve 100,000,000 users from around the world
- What needs to change?

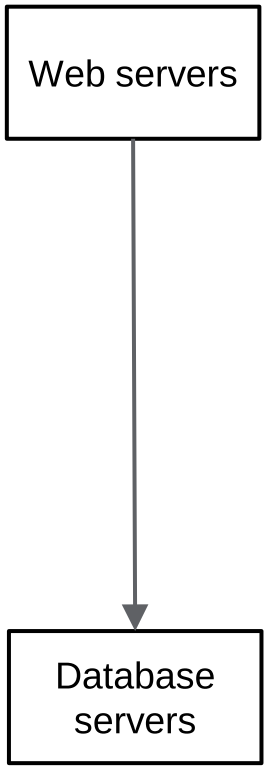
What does 100M users mean



Lots of servers

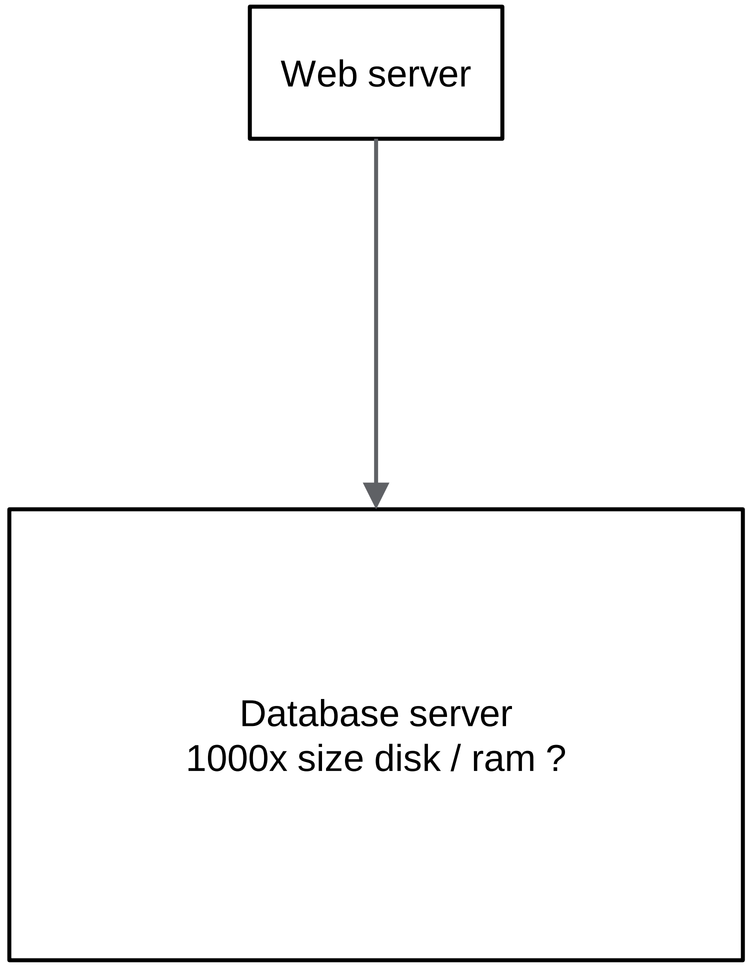

20 M disk Seeks
20,000 disks
833 servers
(24 disk each)
Vertical stack of servers 150m high.
Draws the power of about 900 homes.
~100억원 From Dell.

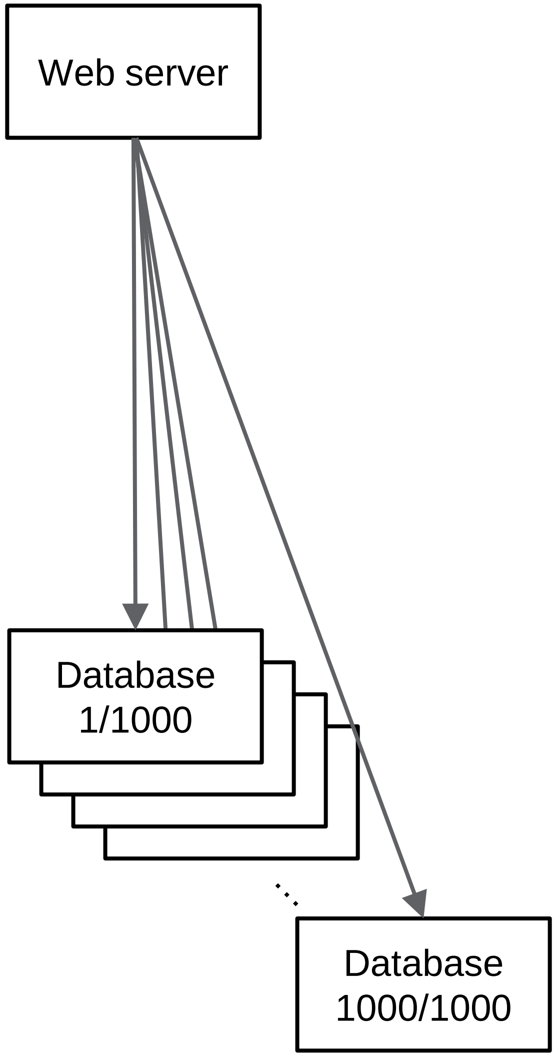

Web server
Web server
Database
Database


Web servers
Database servers

Scale the amount of data

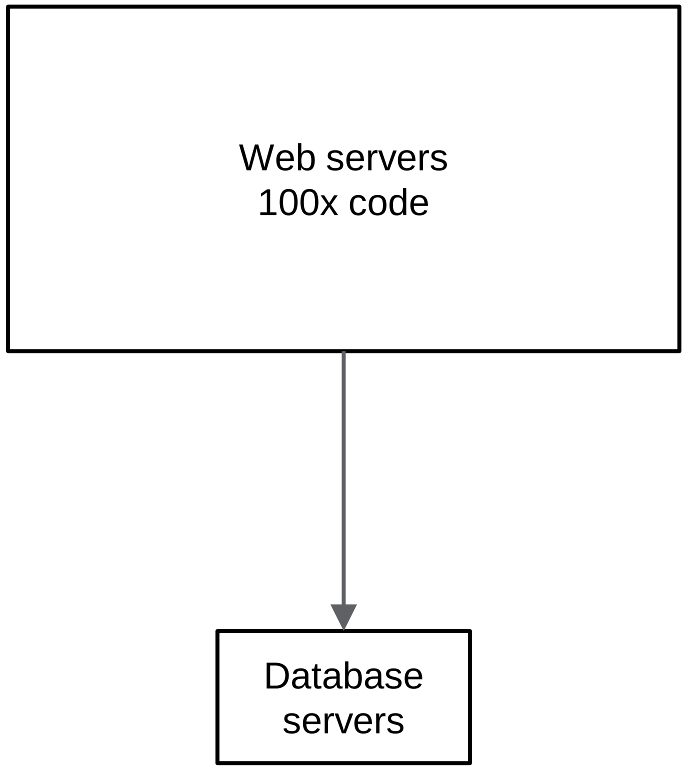
- Now imagine your web application has to serve 1000x amount of data.
- What needs to change?

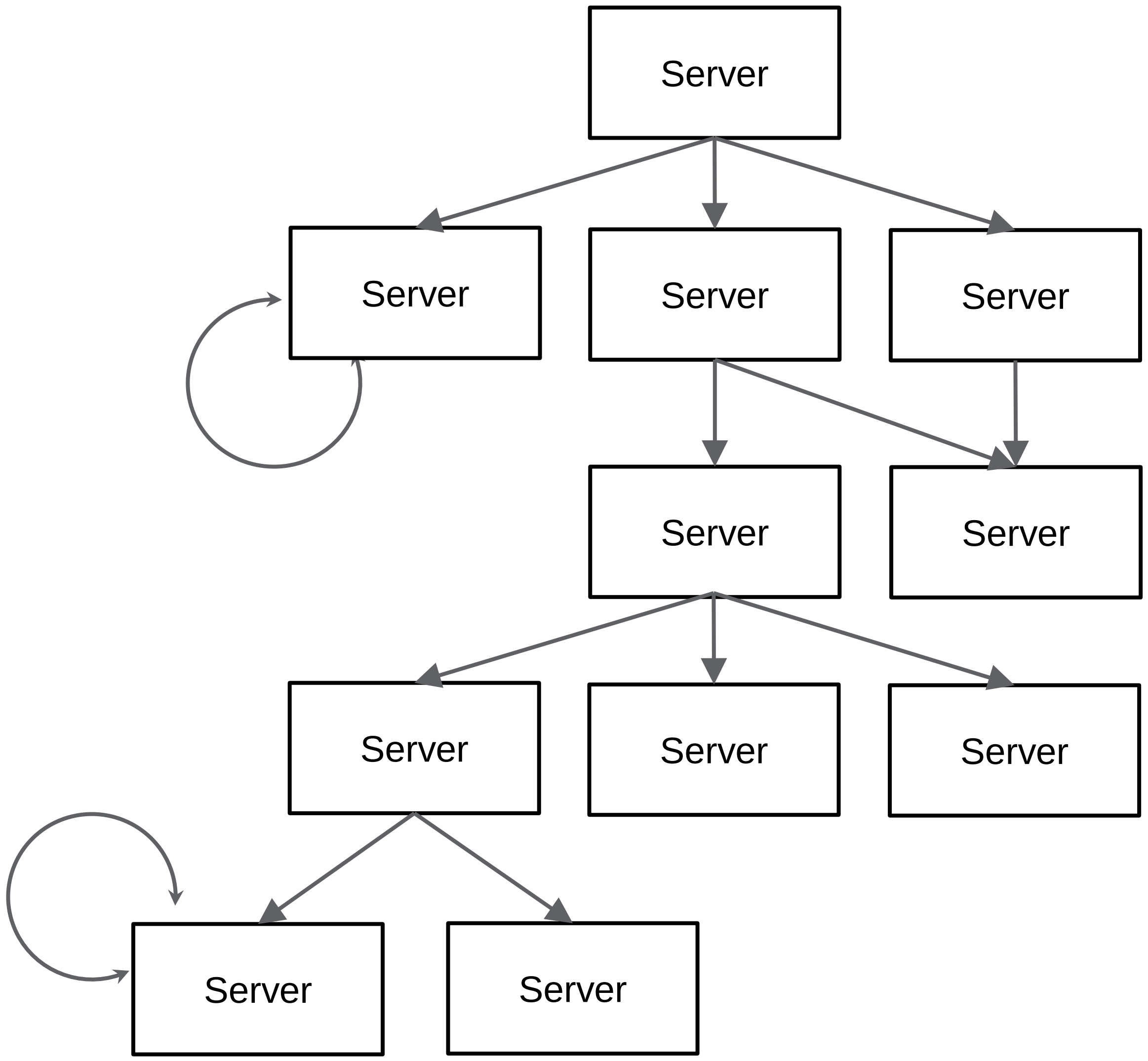

Web server
Database server
1000x size disk / ram ?

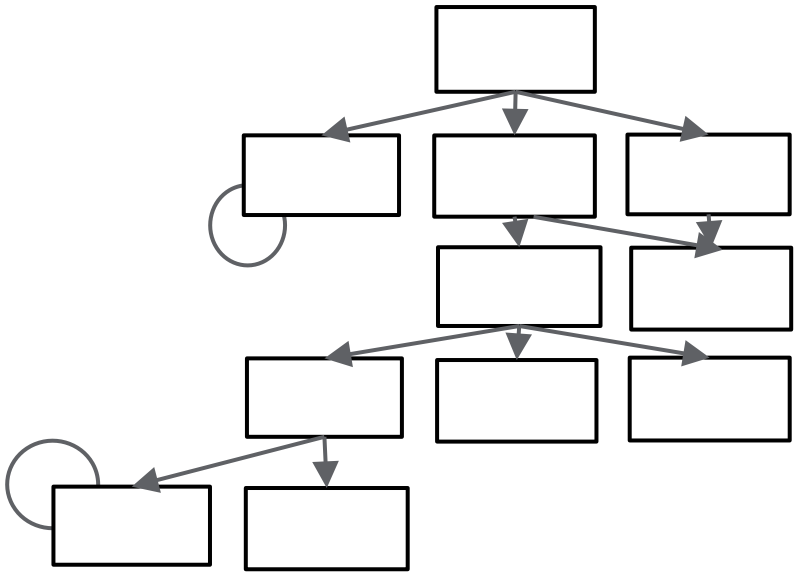

Web server
Database 1/1000
...
Database 1000/1000

Scale the amount of code / engineers

- Now imagine your web application has 100x features, with 100x people working on it.
- What needs to change?

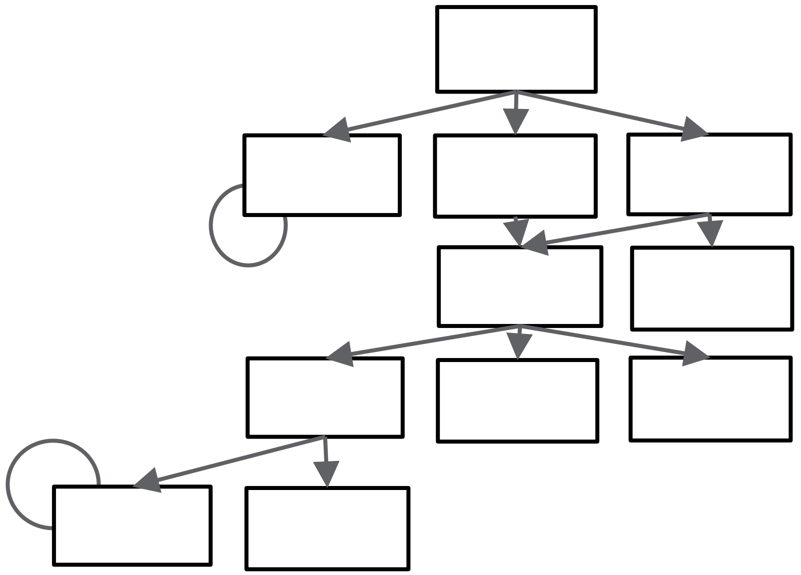

Web servers
100x code
Database servers

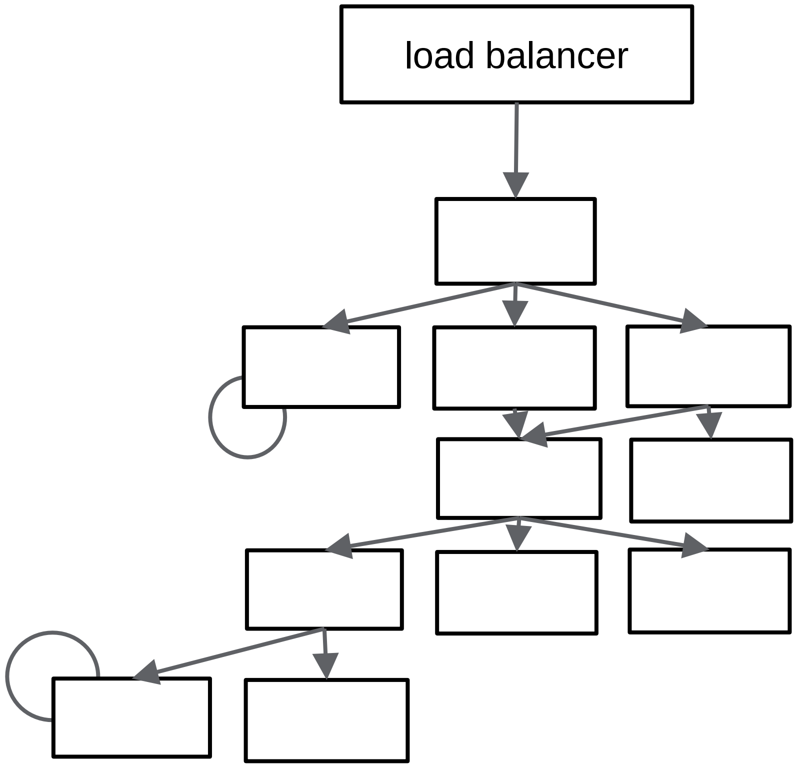

Server
Server
Server
Server
Server
Server
Server
Server
Server
Server
Server

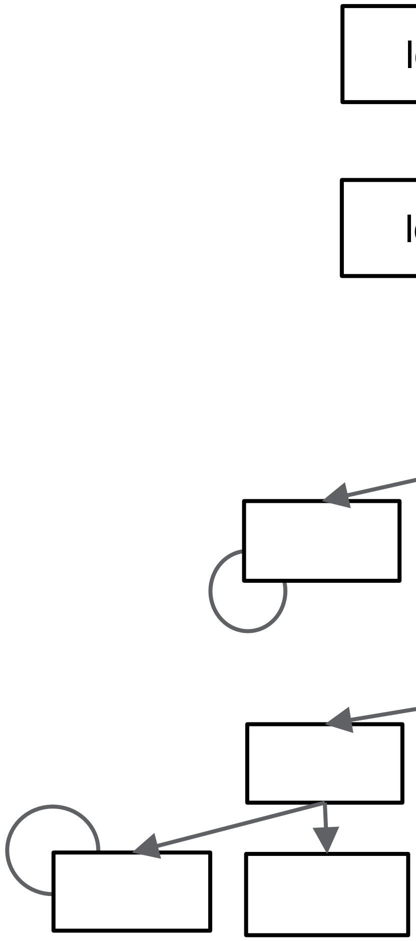


Spread traffic to 10000 servers?

- Don't want to publish 10000 IP addresses to the world
  - IP addresses are scarce and expensive
- Need a frontend loadbalancer to map few IP addresses to a lot of backend servers

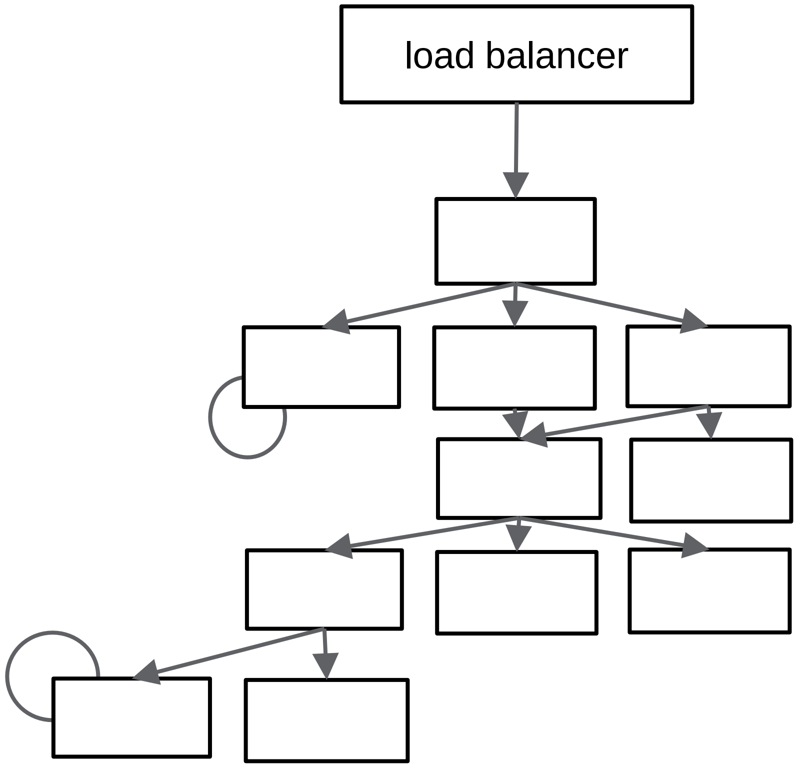


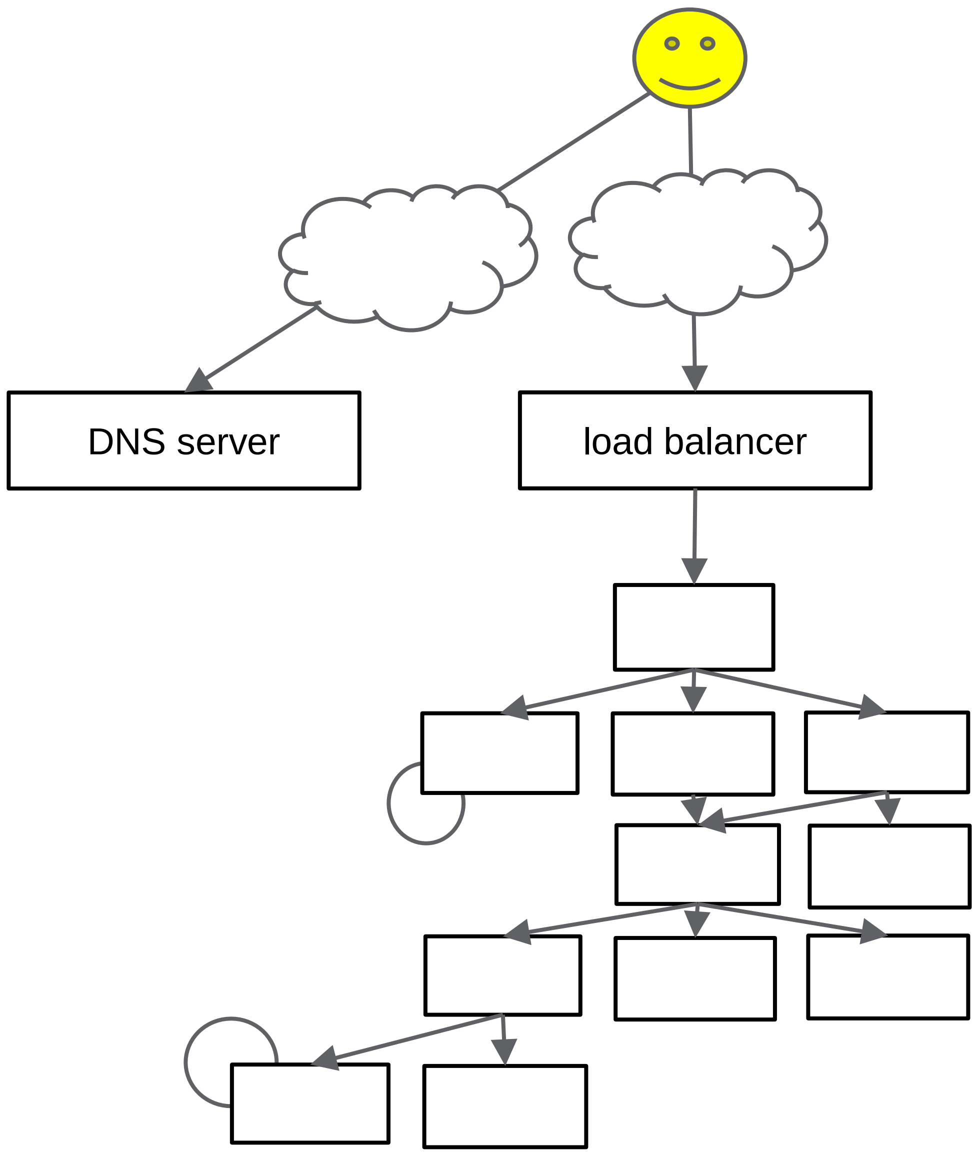

load balancer


load balancer
load balancer

Clients to our loadbalancers

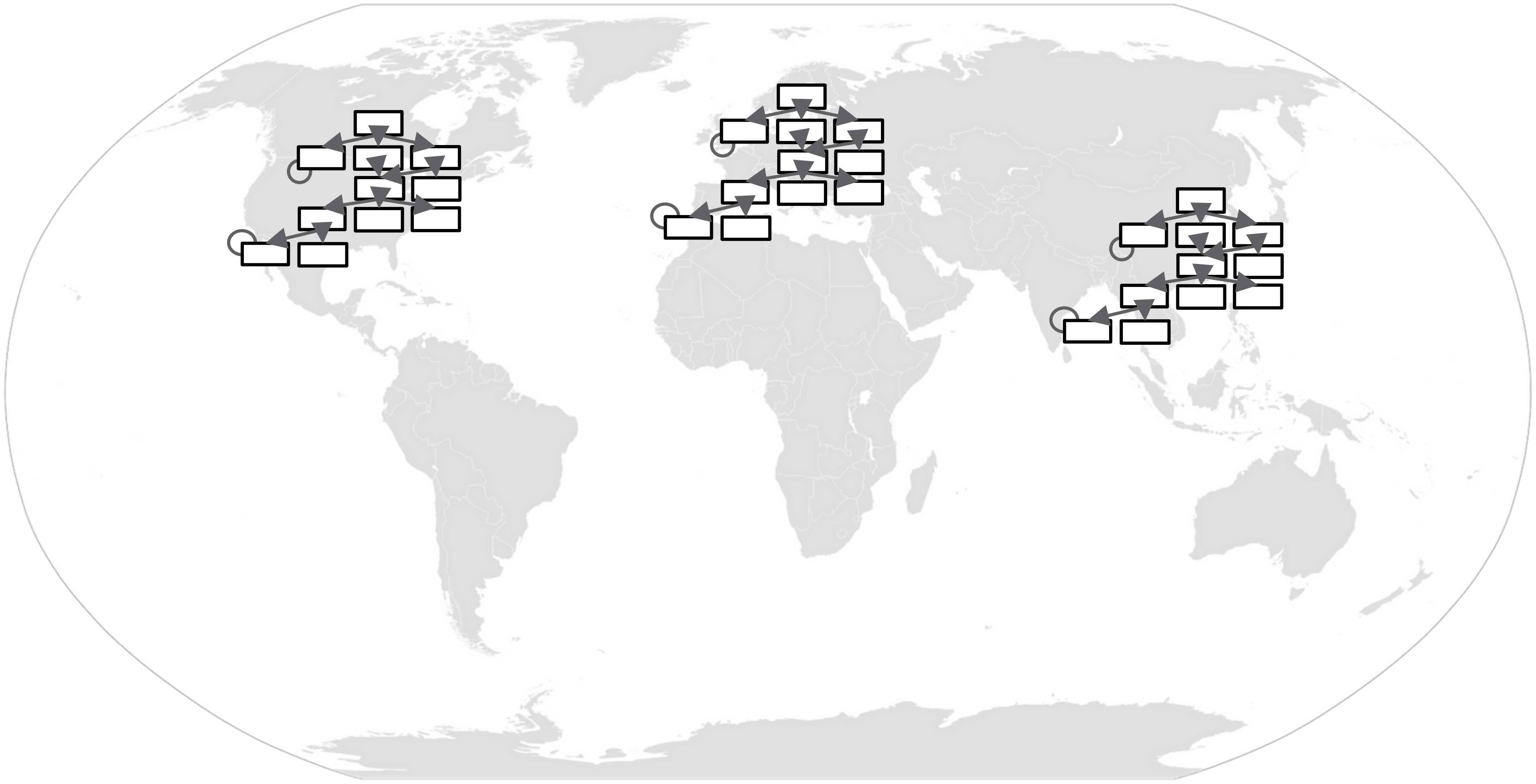
- Clients find frontend loadbalancers via DNS
- Need a DNS server as well


load balancer


DNS server
load balancer

Scale across the world

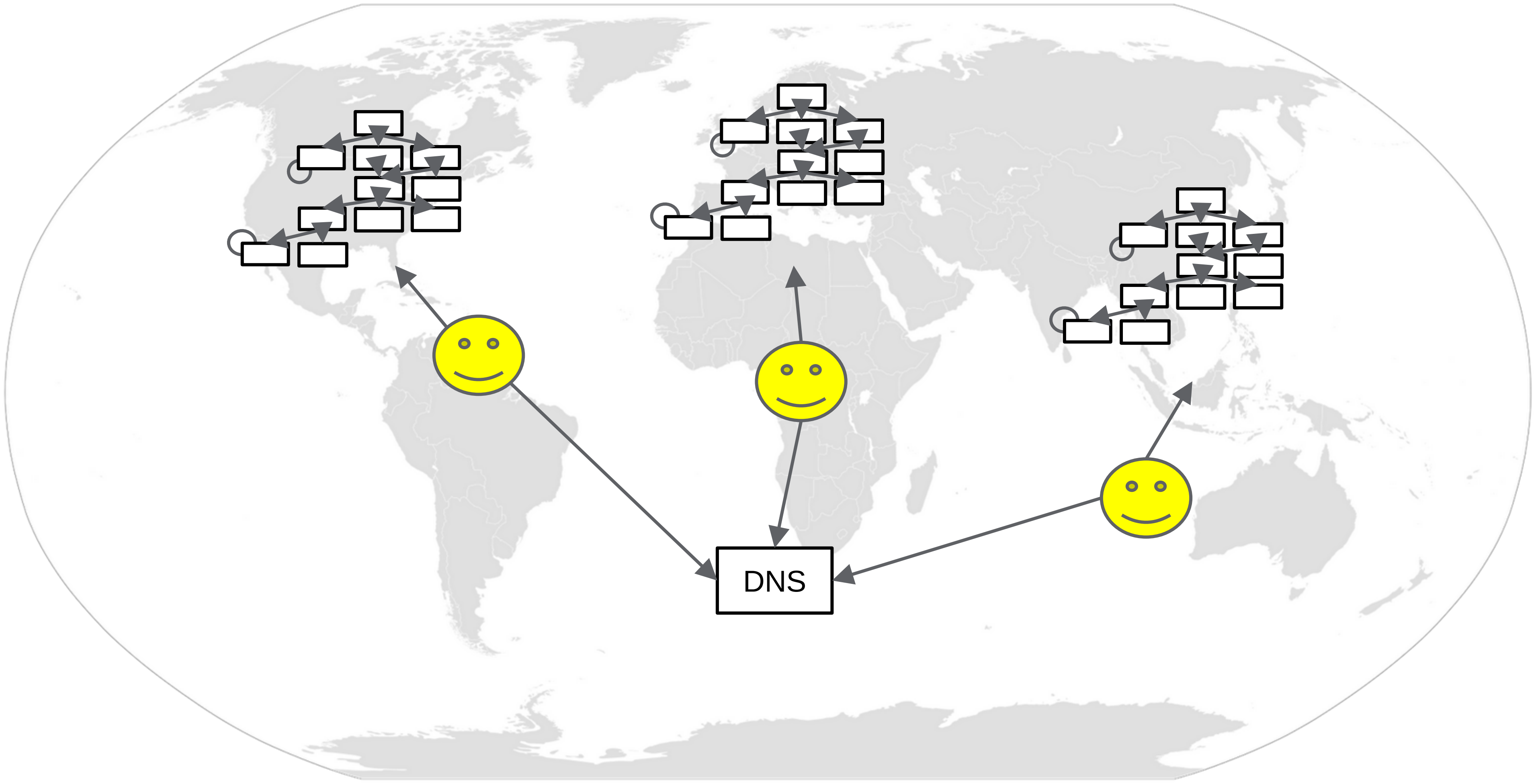
- Users in all continents, geographies.
- Earth is big. Speed of light isn't infinite.
- RTT across
  - Pacific: 125 ms.
  - Atlantic: 90 ms.
  - North America and South America: 120 ms.
  - Sidney to Taiwain: 100 ms.
  - US west coast to east coast: 80ms.

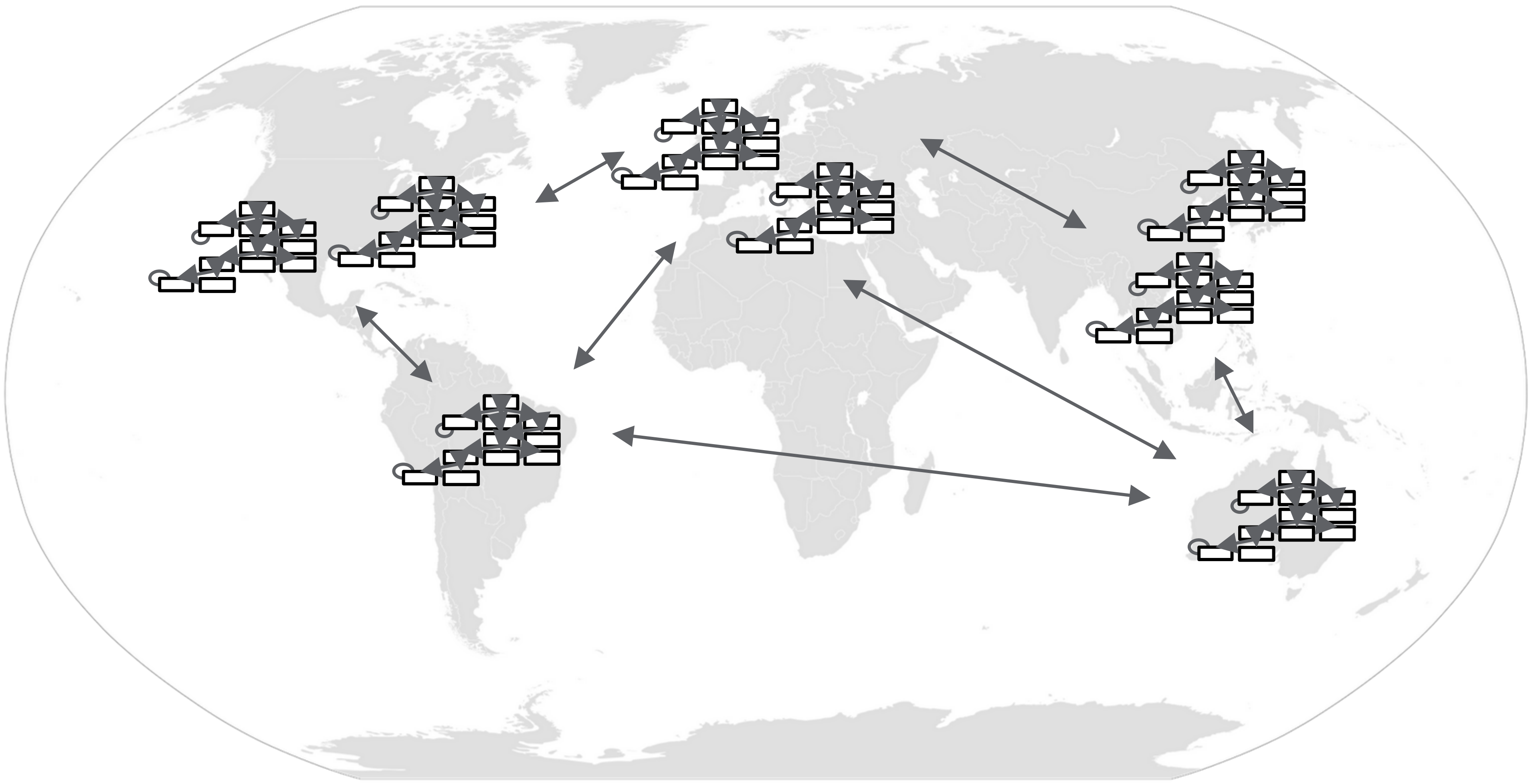


DO COOL THINGS THAT MATTER

Get traffic to the right datacenter?

- Now we have a new problem
- Figure out where the user is
- Send them to the closest datacenter
  - we don't want to send a user in Japan to a datacenter in Europe if we have free capacity in Japan
- "geographic loadbalancing"


DNS

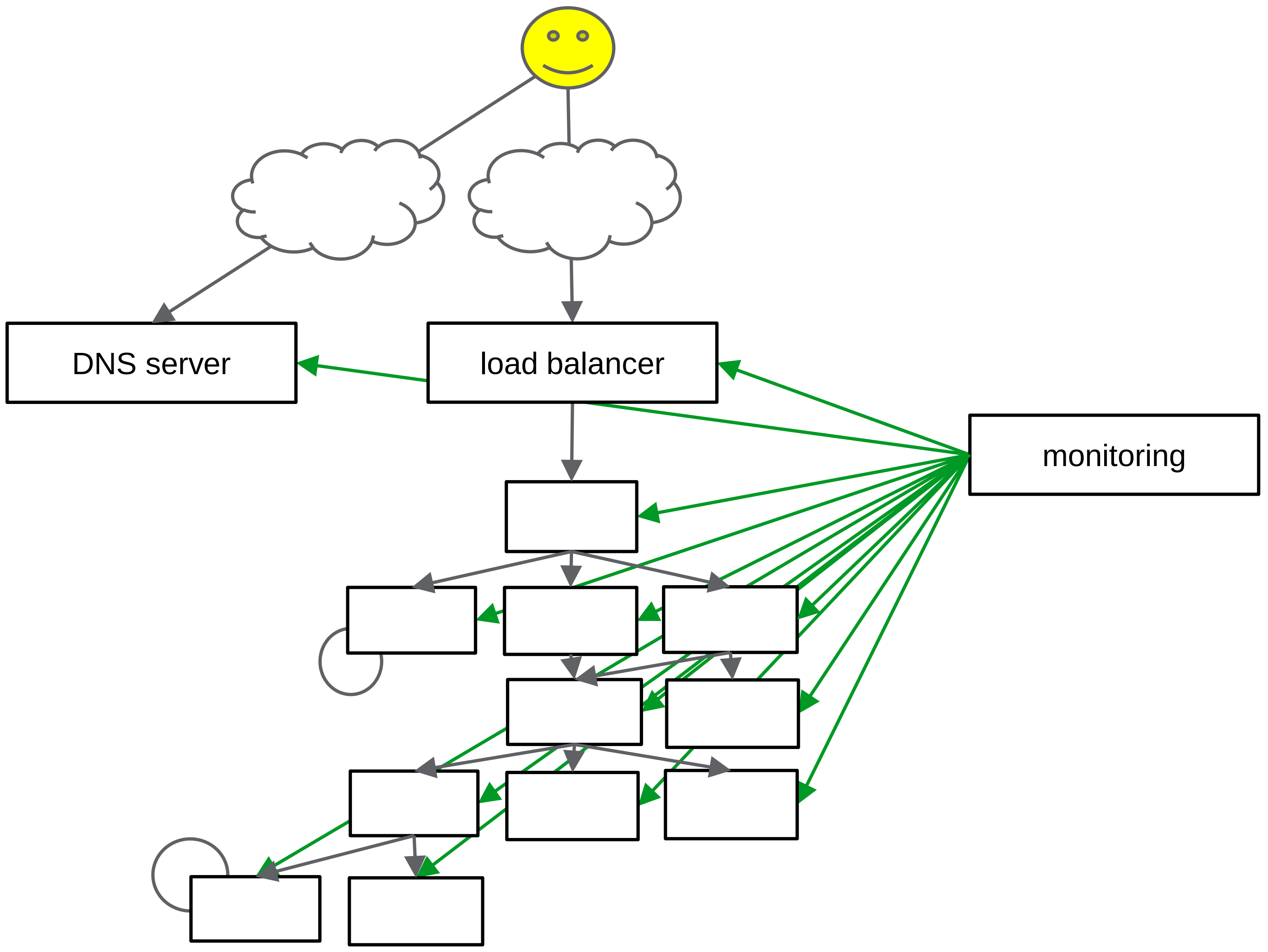


Failure always happens

Failure also scales :(

- 100k separate servers
- Any one problem is a needle in a haystack
- Human can’t keep an eye on all of them, nor act on them

- System has to self-monitor and self-heal
- Debugging and maintenance become a data analysis / data science problem
- Monitoring to figure out what the servers are doing
  - How loaded are they?
  - When are they getting overloaded?
  - How many requests are they handling?
  - Are they providing the functionality we expect?
  - What problems are happening ?
  - How are users being affected ?


DNS server
load balancer
monitoring

Everything can fail

Murphy's Law:

Anything that can go wrong, will go wrong.

- Things that fail:
  - hard drives, memory, cpu, network card, flash memory
  - power supplies
  - switches
  - routers
  - fibre lines
  - power substations
  - any software system (firmware, bios, kernel, os, application)
- Deal with all of these scenarios and beyond

Failures at Scale

“In Computer Science, the rare event always happens, and the impossible happens occasionally.”

- Suppose 5% failure in 5 years in *one* component.

100k machines means 5k failures. 2.7 per day.

- There’s ~10-50 such components. Failure happens always.
- Bathtub curve

Redundancy

- Traditional approach to reliability: If outage of one component causes problems, use two of the same and hope they don't fail at the same time.
- Applicable to a lot of problems: redundant power supplies, drives (RAID), networking (redundant switches, routers etc.)
- Redundancy is expensive, typically 2x
- Holistic approach:
    - Optimize total reliability of service globally

Double failures

Finagle's Law of Dynamic Negatives:

Anything that can go wrong, will -- at the worst possible moment.

- Thanks to scale, double failures are not rare.
- Defense in depth.
- N + 2

Failure domain: Machine

- Example: PSU MTBF is 100,000 hours
  - 100,000 machines mean one PSU will fail every 1 hours
- Machine failure symptoms
  - A single machine suddenly goes offline
  - Or gets slow
  - Or starts corrupting packets
  - Or ... (insert any known symptom here)
- Action: get traffic away from the failing machine
  - and get somebody to repair the machine
- If the affected machine holds any data, need to make sure that data is also available somewhere else.

Failure domain: Switch

- Symptom: dozens of machines connected to one switch go offline at the same time
- Action: get traffic away from affected machines
  - And get someone to replace the switch
- If those machines hold any data
  - And the other location of that data is on the same switch
  - Congratulations, you've just lost actual data

Failure domain: Datacenter

- Some failure modes can take out entire datacenters
  - power outage (across two separate utility power suppliers)
  - hurricanes
  - flooding
  - earthquakes
  - …
- Happen very rarely, but are the hardest to deal with
- Being in just one region/country of the world is not enough
- Need geographic diversity
- Careful when choosing a site

Failure domain: Software System

- Some software systems are global singleton by its nature
  - DNS assignment
  - Top-level loadbalancing system
  - Global master election system
  - BGP / routing
- Happens very rarely, but impact is almost always global

Deal with failure: Divert traffic away

- First we wanted to get traffic to our machines
- Now we want to get it away again!
  - Because machines fail
- Need to figure out when to divert traffic away
  - Monitoring
- Can use the same mechanisms that we used for getting traffic to machines
  - see previous slides about loadbalancing

Disaster Recovery

- Some failures are “disasters”
- Examples
    - One of your major datacenters burns down
    - A software bug silently corrupts your data over the course of months
- Not an option: "oops, we didn't think of that"
- Prepare emergency scenarios
    - how to bring up your service somewhere else when the datacenter burns down
    - how to get your data back when it gets corrupted or lost
    - offsite backups

Beware of mitigation

Cascading failure: Overload

- Even with careful planning, you can and will run out of capacity, globally or locally
- Many systems don’t deal with overload condition well
    - queueing delay shoot through the roof - cpu, network, hard disk
    - backed-up requests pile up in memory
        - blowing up cache, slowing system down
        - possible out of memory
    - network gets saturated, starts dropping packets, tcp retries…
    - modern cpus overheat, kicking thermal throttling
- If one cluster gets overloaded and fails, global loadbalancing system may divert all that traffic to the next cluster, which knocks it over, rinse&repeat

Cascading failure: Crash

- Imagine a crash bug in the software not caught during testing
  - can be triggered by user action (“query of death”)
  - can be triggered by some state change at the service
- Users unaware of such problems can repeatedly attempt to use the service, killing the servers
- When a datacenter failures, loadbalancer will fail over to the next cluster, the user (or users) tries again, knock it over, repeat.

Cascading failure: Self-protection

- Graceful degradation
- Adaptive self-protection mechanism
    - against running out of memory, cpu, disk, network
- Feedback loop creating oscillation

Scale begats problems

Long tail latency

- A single machine can’t handle all data, nor can it do fast enough.
- “Shard” the data - divide it across N machines.
  - Parallelize the access.
- Buffering at NIC, switch, packet loss, kernel thread scheduling, context switch overhead, thread migration, mutex contention, cache miss, TLB miss, queueing delay, imperfect loadbalancing
- As N goes up, 90th, 99th, 99.9th latency go up dramatically
  - But we want 99.99% success within X ms !

Control tail latency

- Drop slow machine
    - When you don’t need all data, don’t wait for the slowest X %.
- Replicate data
    - Try all replicas, and pick the fastest.
    - Hedge: Try one replica, wait Y ms, try another replica, repeat.
- Adaptive loadbalancing
    - Slower machines get less requests
- Internal timeslicing
- Multiple queues
- Avoid network hotspots
- Cache at multiple layers

Too much success

- More users are better
- But too much of a good thing can be a problem
- Need to plan for launch spikes
  - fuelled by press releases, TV news coverage, blog posts etc.
  - or by external events (hurricanes, olympic games,...)
- How do you plan for too much success?
  - borrow machines from somewhere (or purchase time on virtual machines, eg. Amazon EC2 or Google Compute Engine)
  - turn off non-essential functionality and repurpose the machines that supported it
  - deliver degraded results (eg. switch to a cheaper algorithm)
  - limit sign-ups

Change management

- Downtime is not an option
    - Google services are global, thus need to be always up.
    - There is no "scheduled maintenance" for most services.
- Make all changes "in flight"
- Or, take *a fraction* of the service down
    - but that means you lose redundancy
    - and that means you need more redundancy to start out with
- Think about compatibility
    - some fraction of servers might have a change
    - others might not
    - they still need to talk with each other
    - and users need to be able to talk to either the old or the new server
    - backwards and forwards compatibility needs careful planning in advance

Human reliability

Business continuity

- Also called the "bus factor"
    - can you continue running your business if person X gets run over by a bus?
- New people join the team, old-timers leave
- Everybody needs to do their share of emergency response
- Make the systems easy and safe to use
    - even if you don't understand 100% of their ins and outs
- Document the rest

Human mistakes

- Most outages are due to human mistakes or human triggered
- Typo, misleading commands, UI problems
- Design all controls with human mistakes in mind
- In-line documentations are effective. Separate documentations are not.
- Wide communication as early as reasonably possible
- Quick revert
- Blameless postmortem culture

Conclusion

- Failures are difficult to engineer against but not impossible.
- Scale brings its own challenges.
- Careful planning and design choices can go a long way.

Questions?