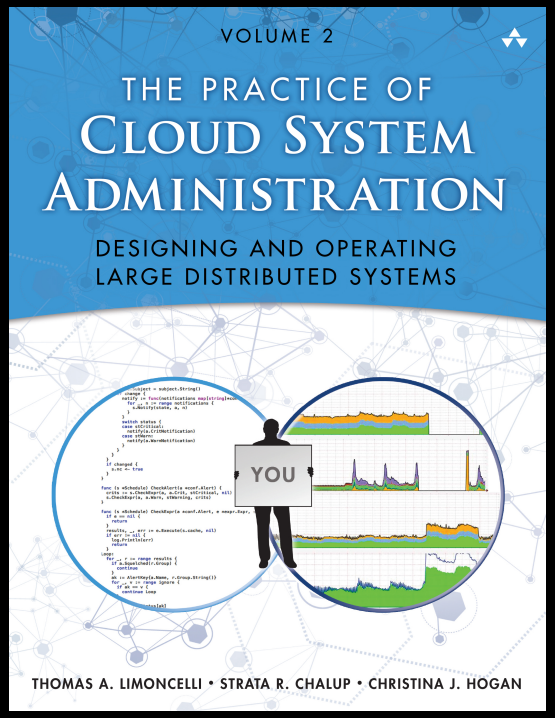
Live Upgrades on Running Systems:

8 Ways to Upgrade a Running Service With Zero Downtime

Tom Limoncelli @YesThatTom
EverythingSysadmin.com

With material from…



Who is Tom Limoncelli?

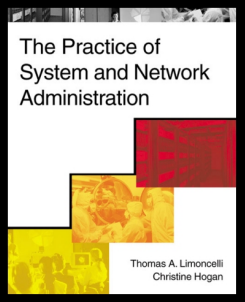
Sysadmin since 1988

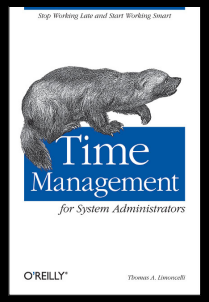
Worked at Google, Bell Labs plus many smaller companies.

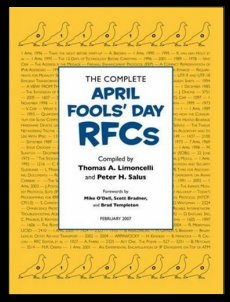
SRE at Stack Exchange, Inc
serverfault.com / stackoverflow.com

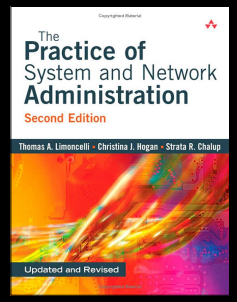
Blog: EverythingSysadmin.com

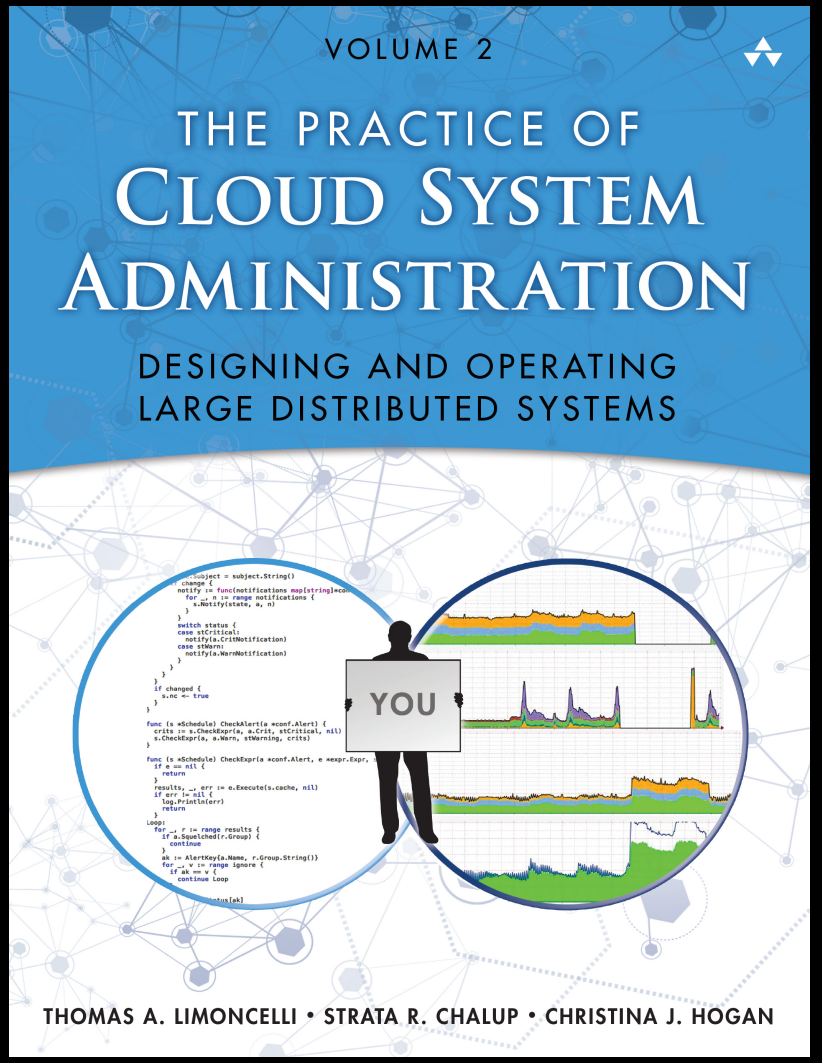
Twitter: @YesThatTom











Goal:

- Zero downtime due to upgrade
- Reduced risk
- Increased confidence

A rose by any other name

- "Upgrades"
- "Code pushes"
- "Deploy"
- "Deployment"
- "Push to production"
- "Software Upgrades"

Why live upgrades?

Why?

- Zero downtime.
- The web is 24/7
- No more "under construction" logo

And...

- If we can do it live, we can do it often.
- More "code pushes" means more ability to innovate.
- ("innovate" -- A big word that means "try new things")

When?

The best time to do upgrades is during the day.

Old View:
Do upgrades at night

- Upgrades require downtime.
- Downtime should be isolated to low-traffic times.
- Risky procedures should be done when mistakes have the least impact.

New View:
Working at night is bad

It is better to avoid...

- Sleepy sysadmins
- Problems happening when key people are unavailable
- Disrupting weekends, sleep, or other leisure

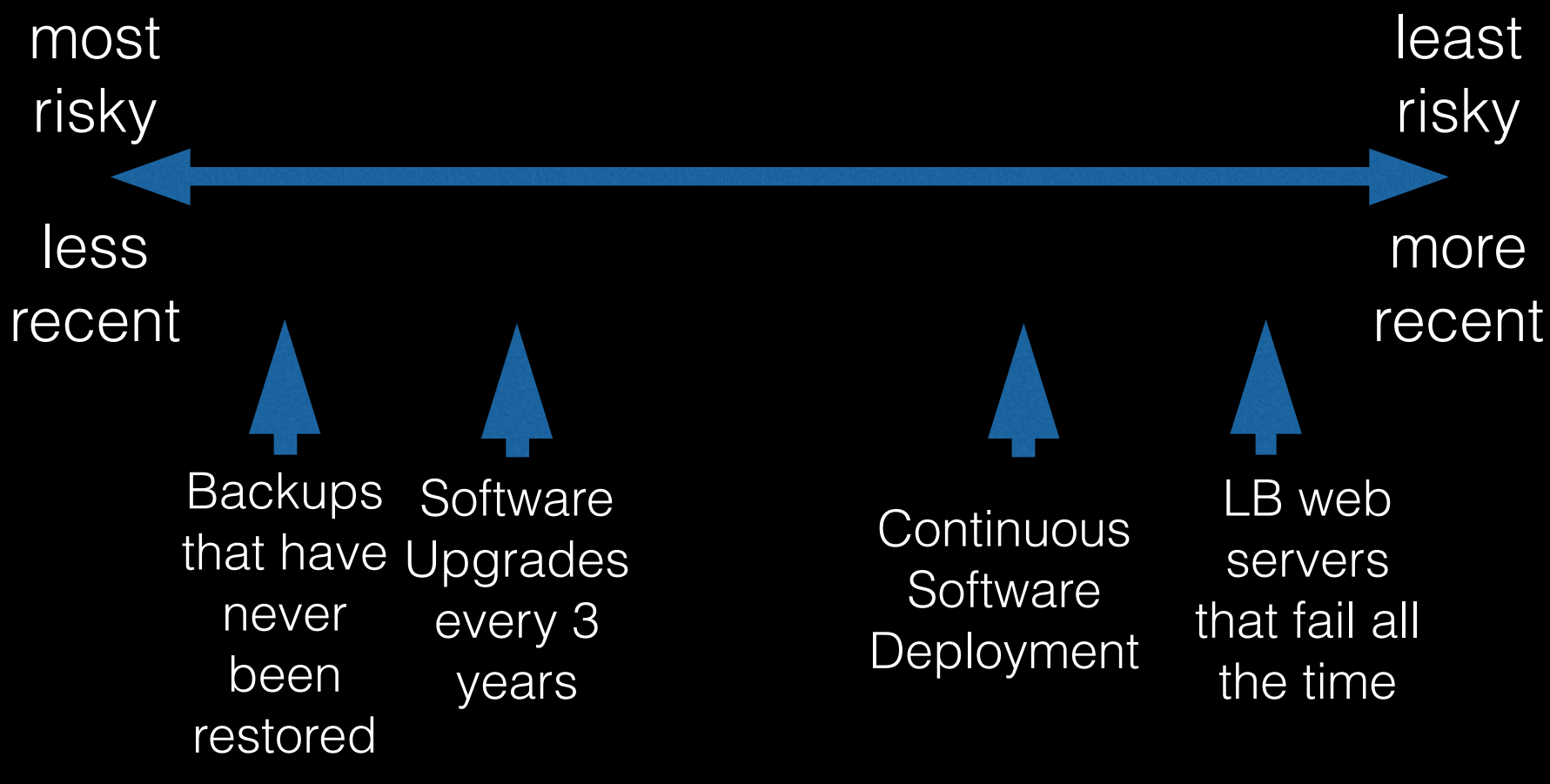
New view:
Day-time upgrades are good

New view:

- Design systems that can be upgrades while running.
- Upgrades done when key people are available, awake, sober.
- Risky procedures should be repeated until they are no longer risky.

Risk is *inversely proportionally* to how recently the process has been used.

Risk is *inversely proportionally* to how recently the process has been used.



Small batches

“Big Bang” releases
are inherently risky.

Small batches are better

Fewer changes each batch:

- If there are bugs, easier to identify source

Reduced lead time:

- It is easier to debug code written recently.

Environment has changed less:

- Fewer “external changes” to break on

Happier, more motivated, employees:

- Instant gratification for all involved

ProTip: Follow the sun upgrades

- Two teams, 8 timezones apart.
- Each team can do upgrades for the other

"Pets vs. Cattle"

Pets are unique, hand reared and lovingly cared for -- they are, by all considerations, members of the family. They have names.

Cattle are largely identical, managed in herds, and bought and sold as a commodity -- they are, in effect, food. Rarely named.

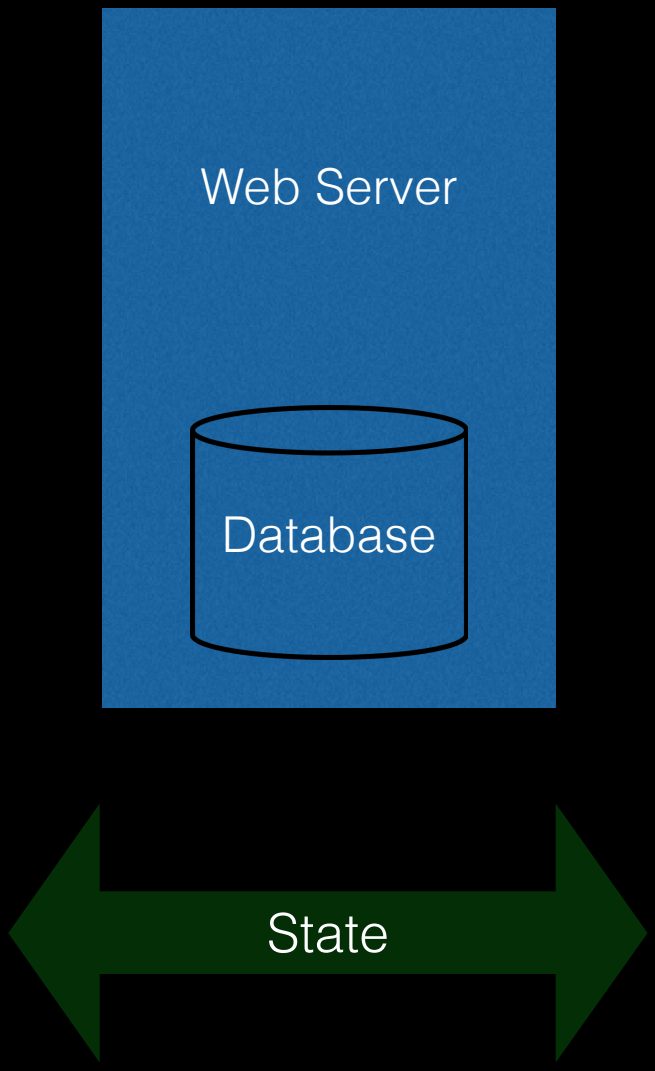
Servers that are Pets

- Carefully constructed.
- If it gets sick, we fix it.
- Example:
  - That enterprise box used for LDAP/SMTP/DNS/DHCP/IMAP

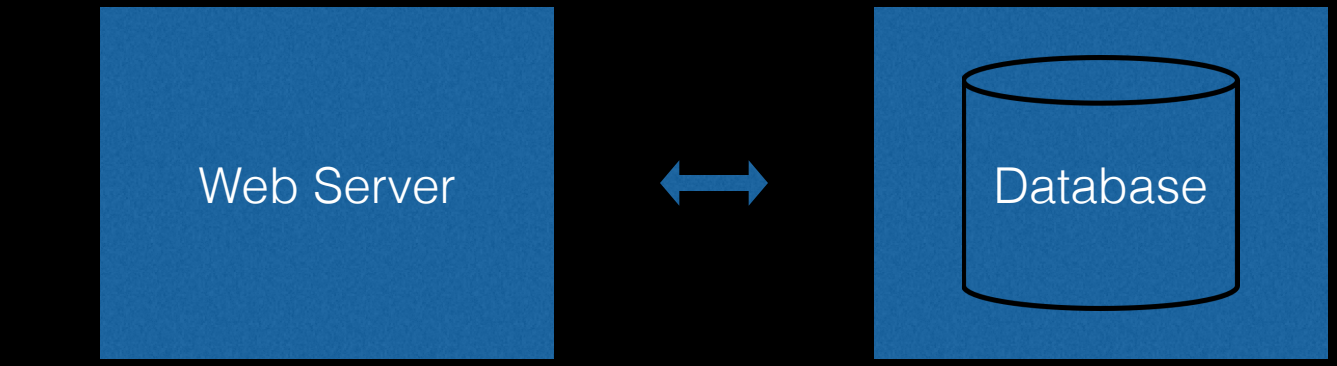
Servers that are Cattle

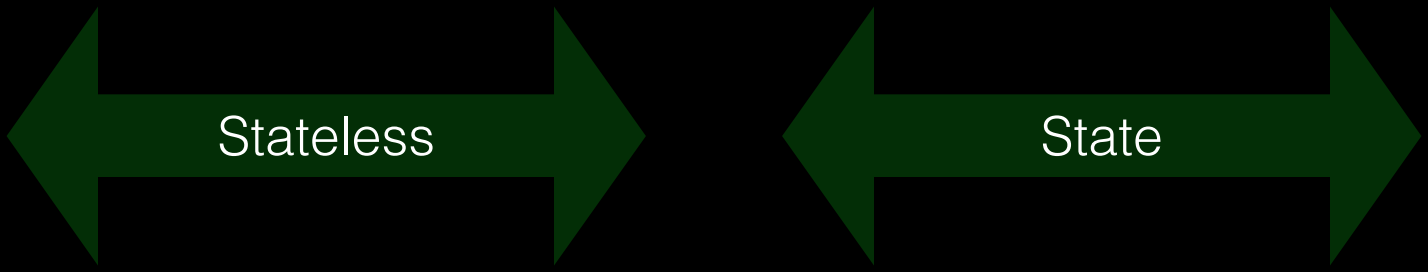
- Automated installation/ configuration.
- Problem? Wipe and reload.
- Example: Web servers with little or no "state".

Pet



Cattle + Pet





Cattle + Cattle + Pet

Homework: reduce the "pets" in your system.

- Try to have fewer pets every year and with each new design.

Technique #0

Upgrade while the service is down

(not a "live upgrade" technique)

UNDER CONSTRUCTION

SITE IS UNDER CONSTRUCTION PROCESS

PLEASE VISIT US LATER

"In-place" upgrade:

- Turn it OFF
- UPGRADE
- Turn it ON

Clone then upgrade clone:

- Clone old machine.
- Upgrade clone.
- Swap into place.

Quiz:
After cloning, do we upgrade
the original or the clone?

Can cloning be done
while the system is up?

Can swapping be done
with no down time?

In-place vs. Clone+Swap

- Less risk
  - You can swap back to original.
- Less downtime
  - But you still have downtime.

"state"

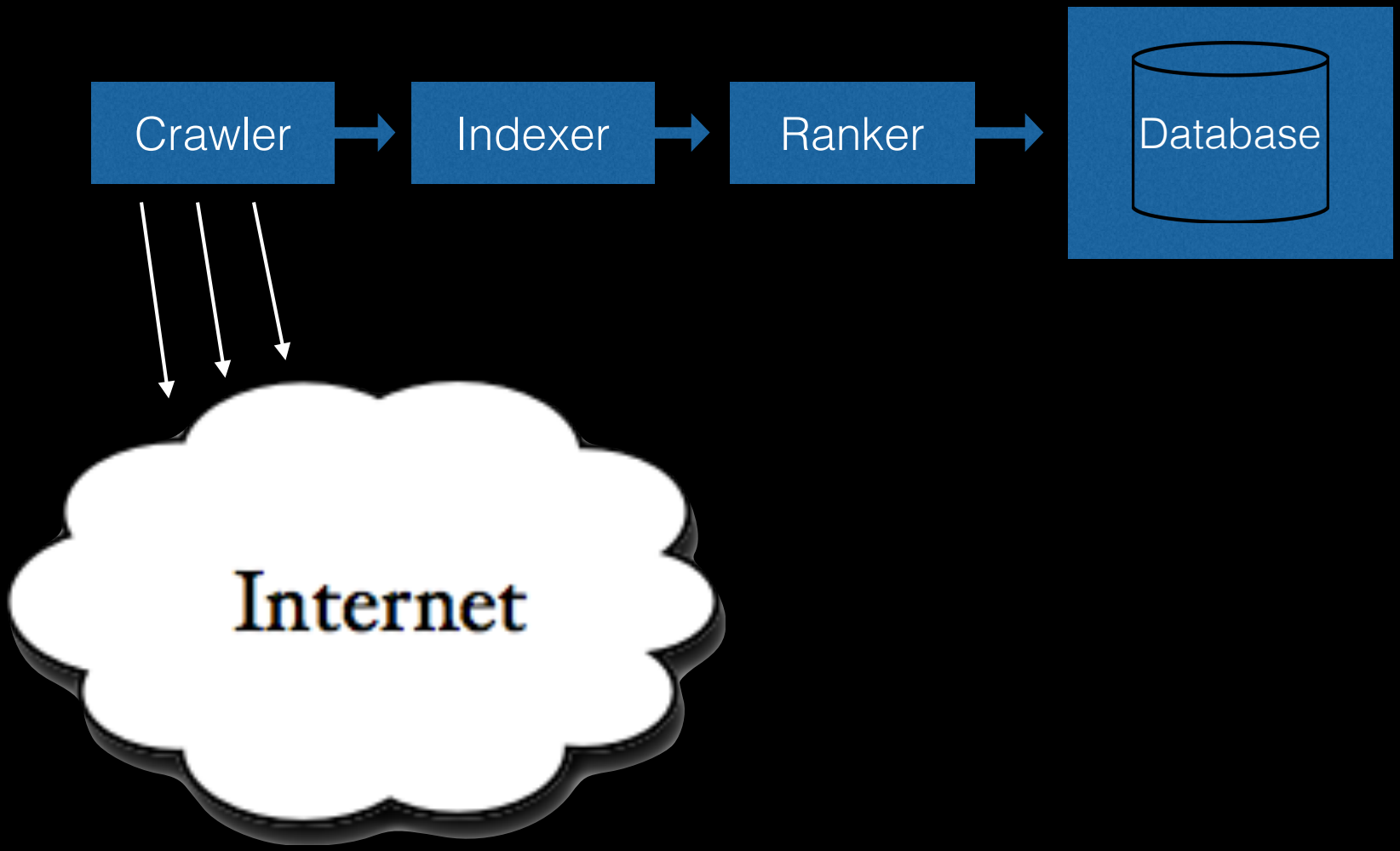
- Databases
- Configurations
- User data
- etc.

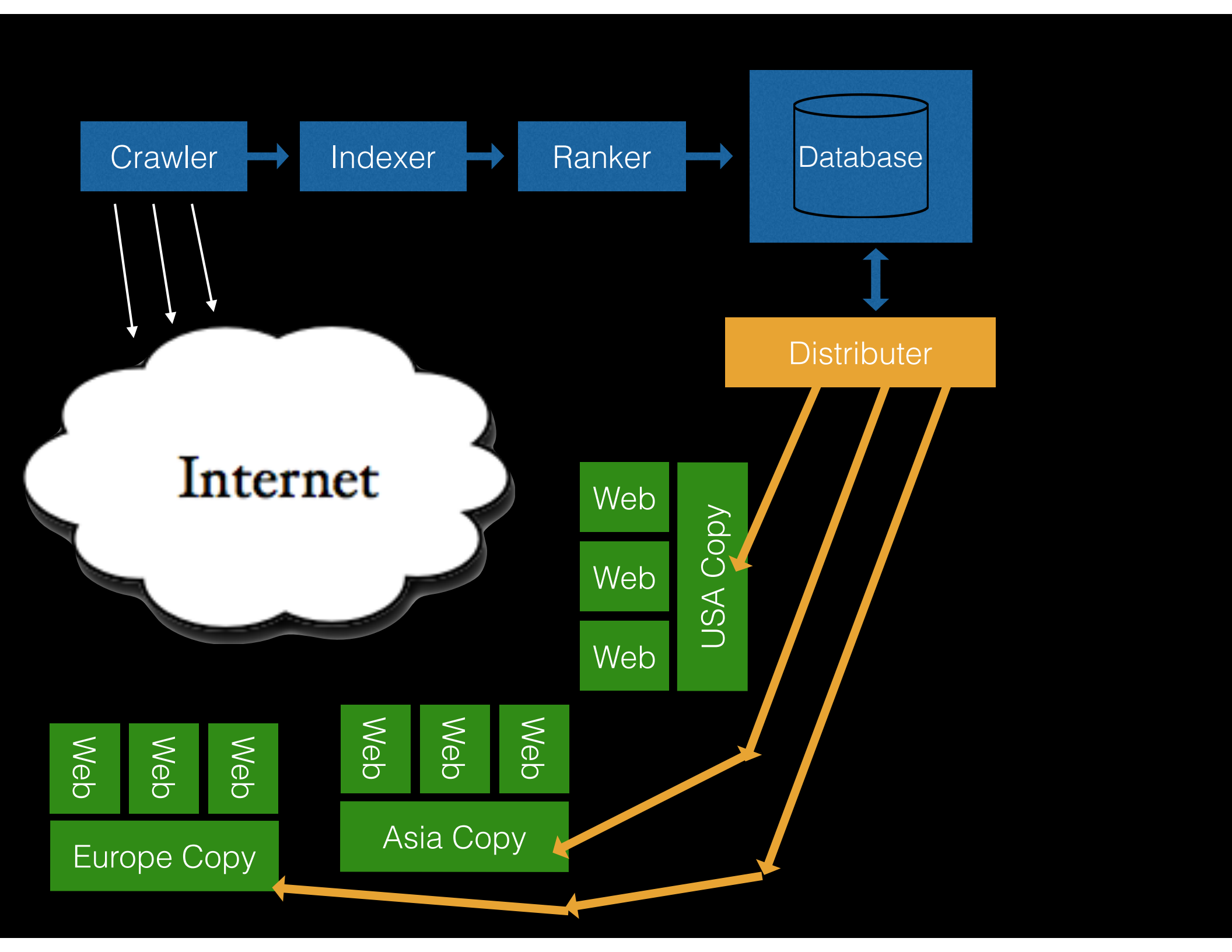
Copying state can take hours, days, months.

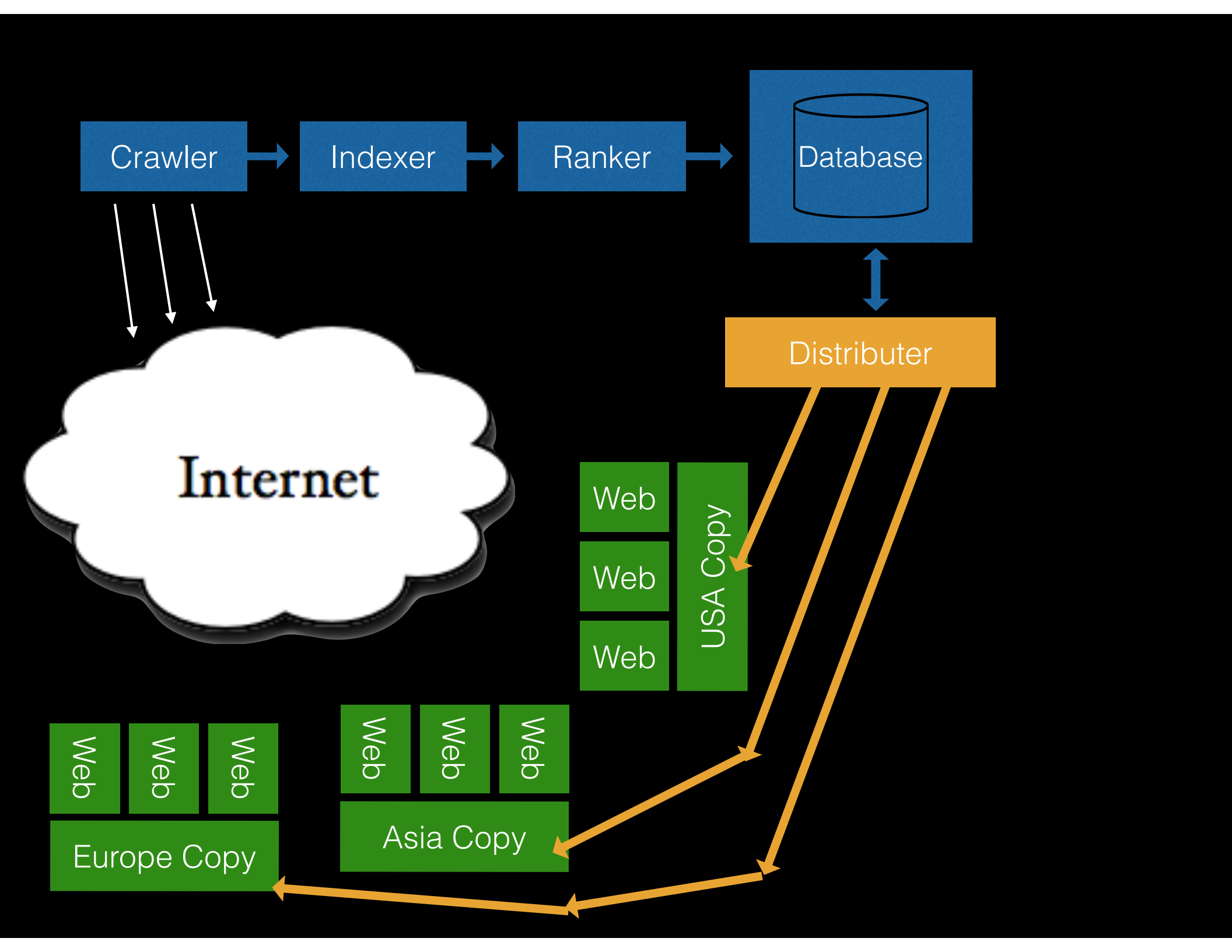
- May be too big to have second copy.
- Copying optimizations: live copy + delta update later

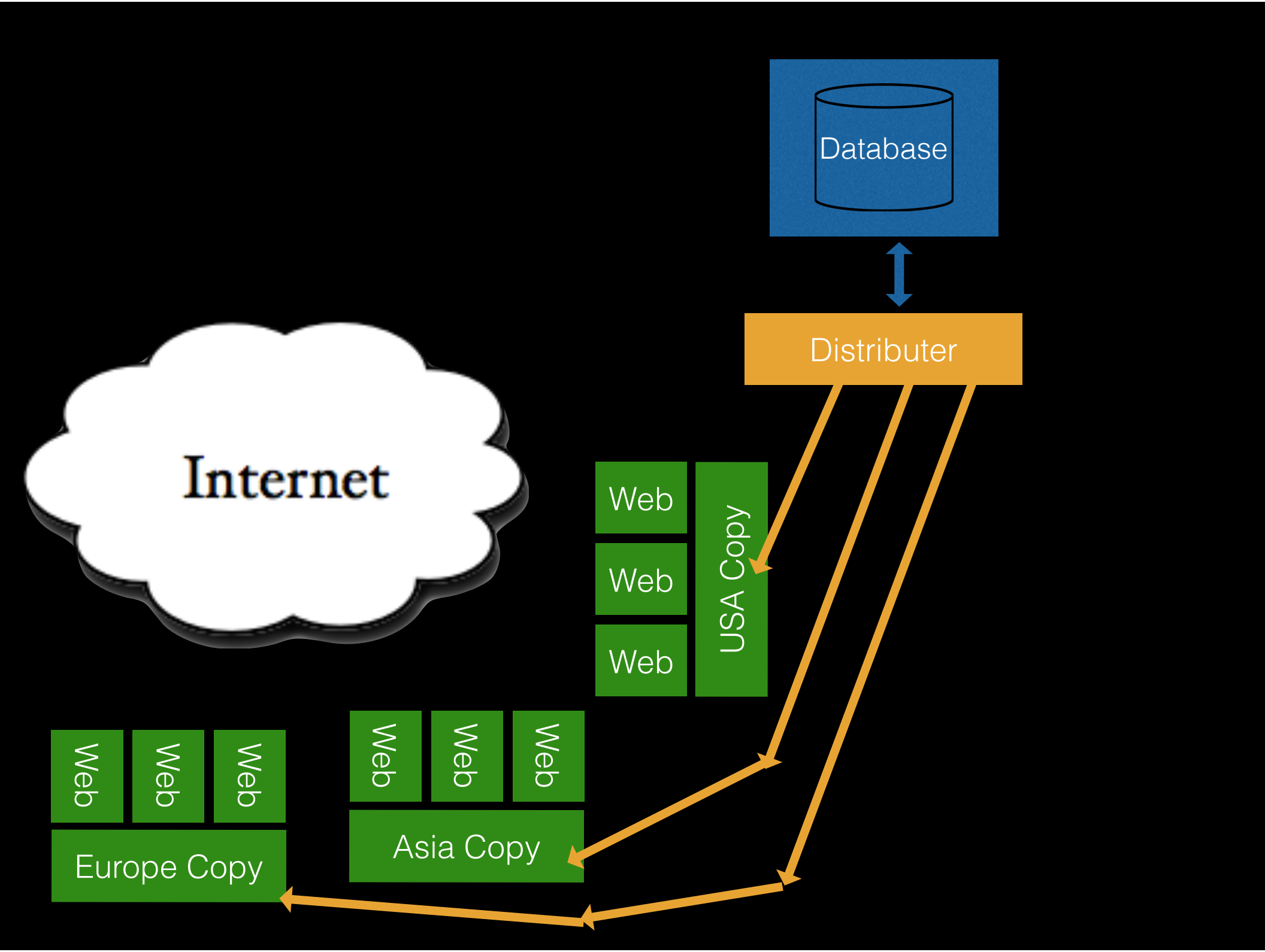
	Risk	Downtime	Copy Time
In-Place	Highest	Medium	None
Clone + Swap	High	A lot	A lot + delta
Install + Swap	High	Some	A lot + delta
Install + Swap w/remote state	Low	Minimal	None

Case Study:
Upgrading the pipeline


Crawler
Indexer
Ranker
Database
Internet


Crawler
Indexer
Ranker
Database
Distributer
Internet
Web
Web
Web
USA Copy
Web
Web
Web
Asia Copy
Web
Web
Web
Europe Copy


Crawler
Indexer
Ranker
Database
Distributer
Internet
Web
Web
Web
USA Copy
Web
Web
Web
Asia Copy
Web
Web
Web
Europe Copy


Database
Distributer
Internet
Web
Web
Web
USA Copy
Web
Web
Web
Asia Copy
Web
Web
Web
Europe Copy

While the pipeline was down

- Duration:
  - The longer the pipeline was down, the less “fresh” the data was.
- But the users never directly felt an outage

When possible, architect your systems to decouple gathering and serving.

Now... 8 live upgrade techniques

Technique #1

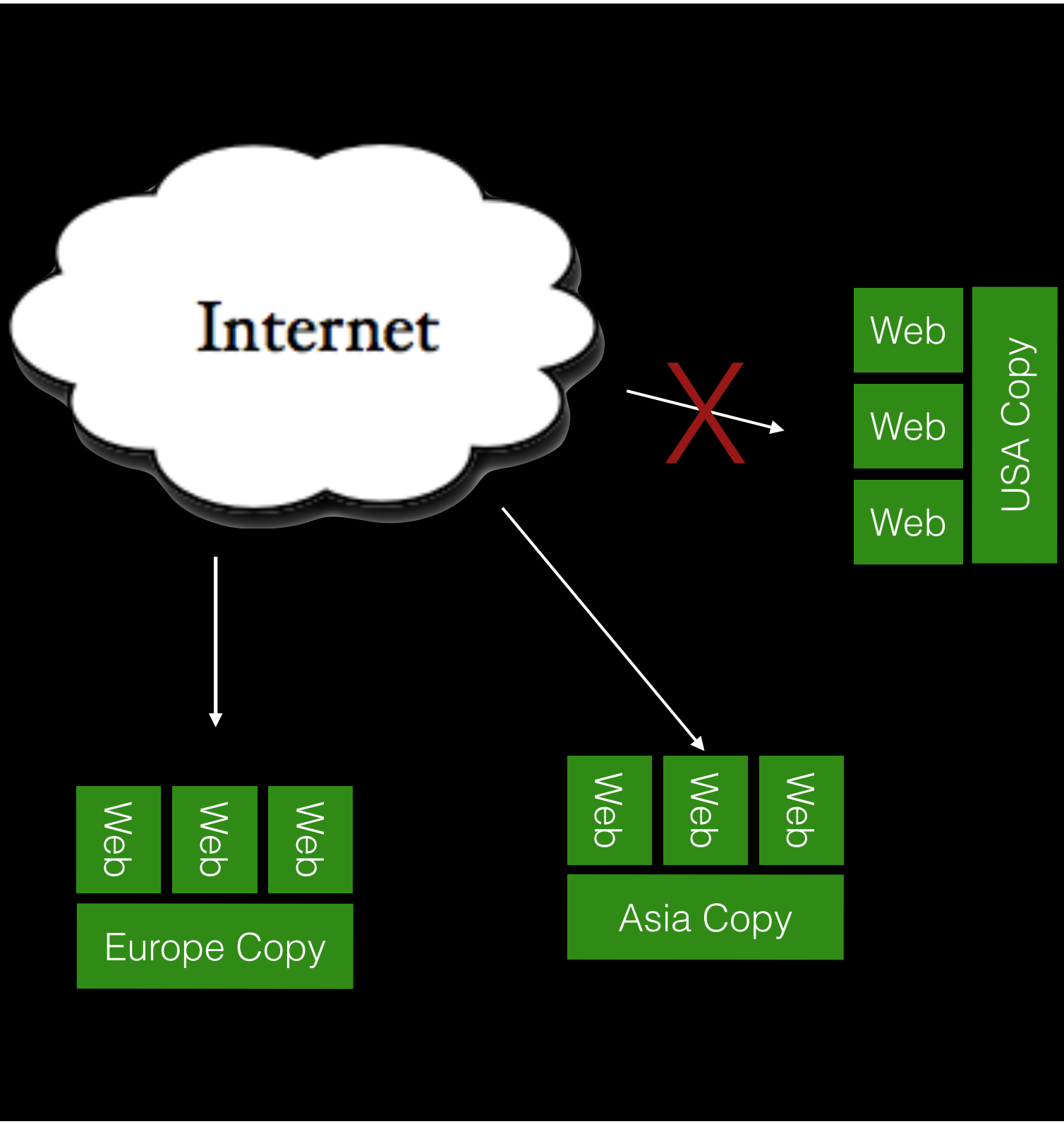
Rolling upgrades

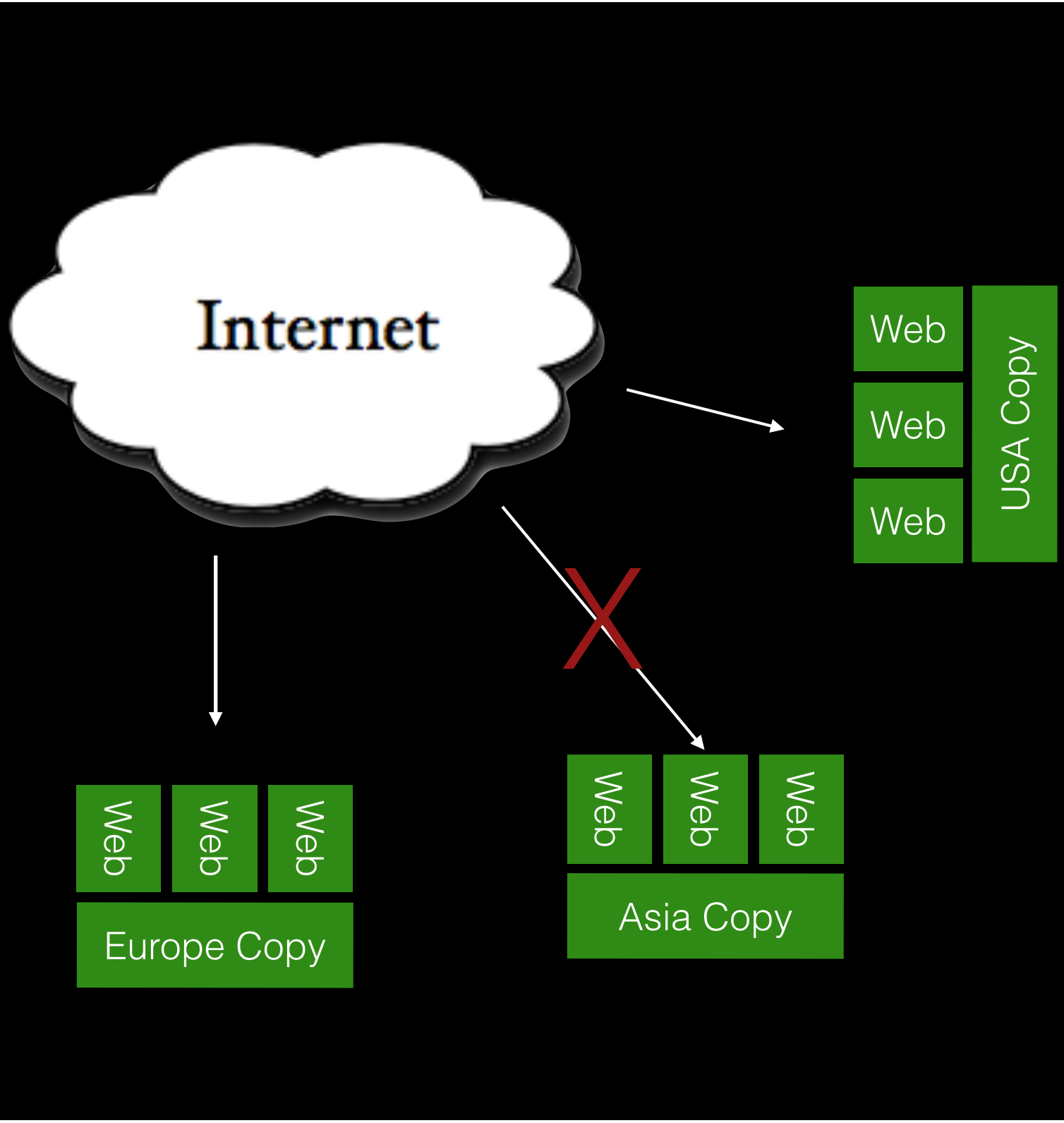
Rolling upgrades

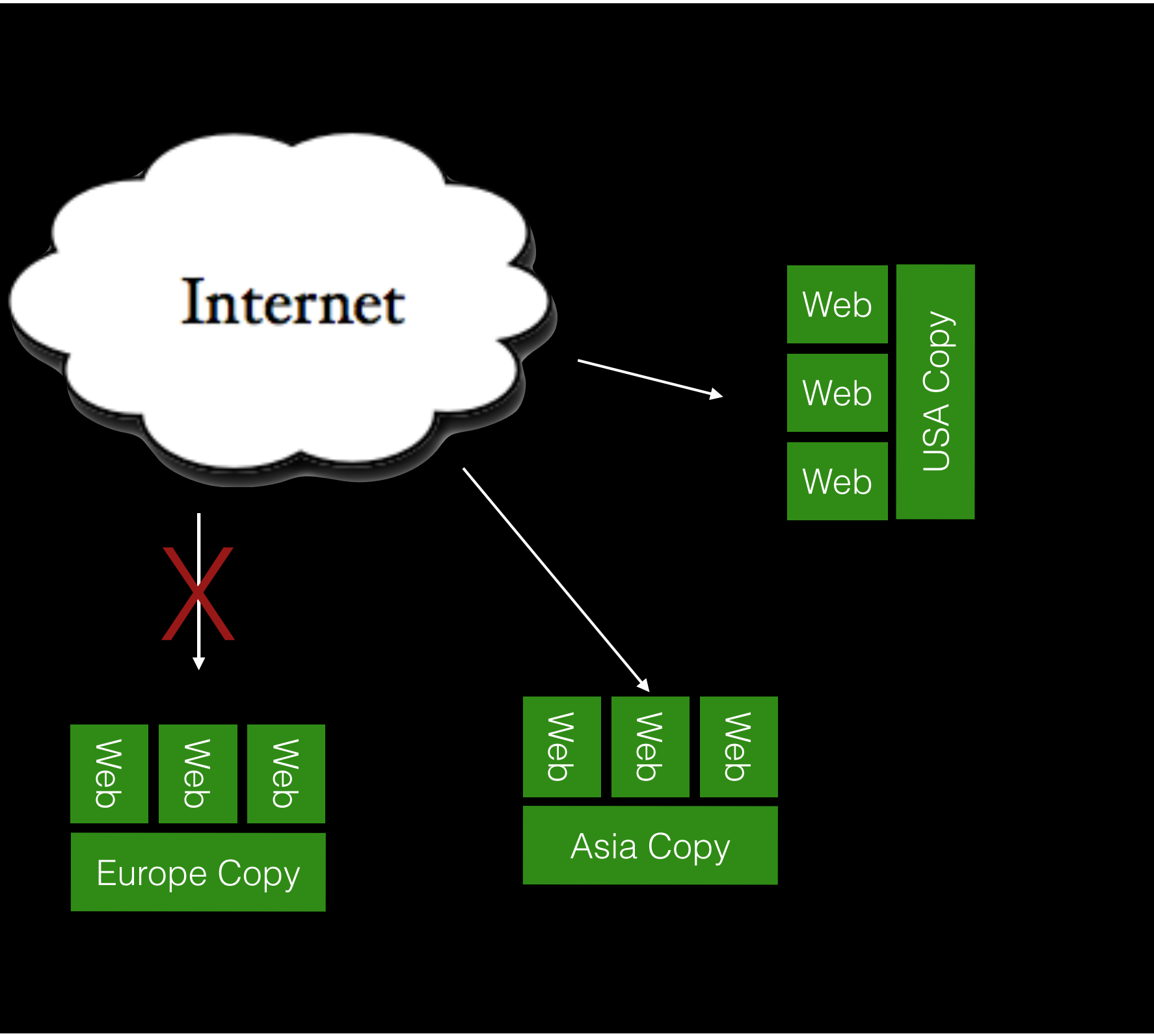
- Upgrade one replica at a time

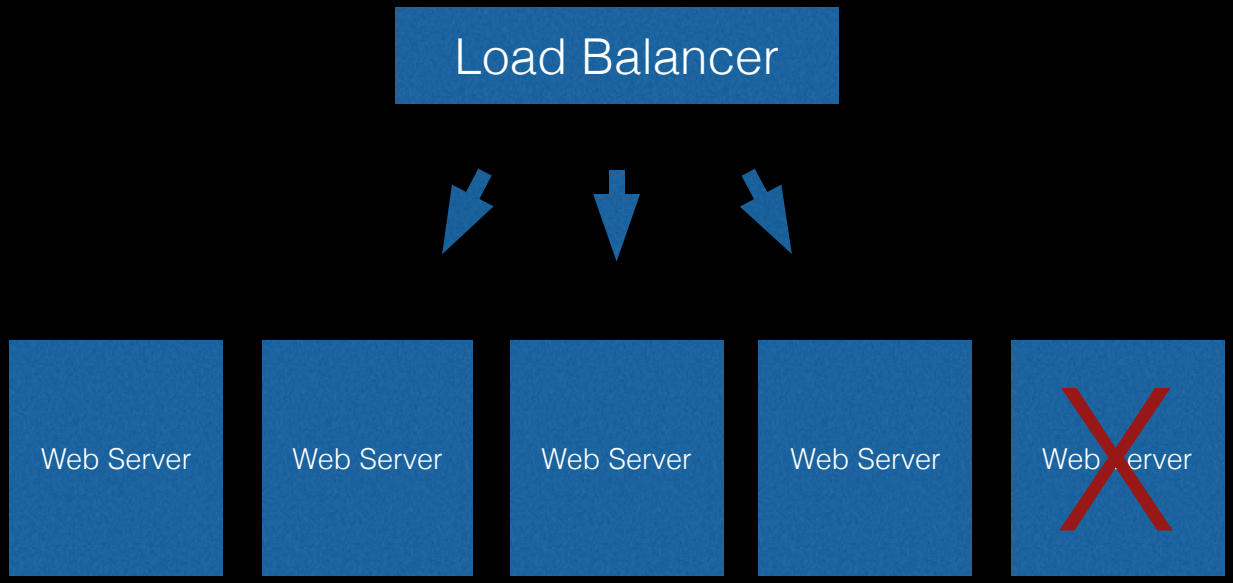
Rolling upgrades

- foreach replica
  - Drain it & Remove from service
  - Upgrade it
  - Test it
  - Return to service
- Repeat


Internet
Web
Web
Web
USA Copy
Web
Web
Web
Europe Copy
Web
Web
Web
Asia Copy


Internet
Web
Web
Web
USA Copy
Web
Web
Web
Europe Copy
Web
Web
Web
Asia Copy


Internet
Web
Web
Web
USA Copy
Web
Web
Web
Europe Copy
Web
Web
Web
Asia Copy


Load Balancer
Web Server
Web Server
Web Server
Web Server
Web Server

What is "Draining"

- Take the item out of the load balancer.
- Wait until no new queries
- Wait until old queries are completed.

Lame Duck Mode

- Programming the LB remotely is difficult.
- Instead, have server "lie" about its health. Claim to be down, but process queries.
- LB will take the server out of rotation.

Starting in lame-duck mode

- Server should not claim to be healthy until it is ready to receive requests.
- Optionally, stay in lame-duck mode until trigger.

Caveats:

- Reduced capacity during upgrades.
- Can remaining machines handle the load?

What if there is an outage?

- ProTip: Have enough capacity for 2 replicas to be down at any time.
- One down for planned maintenance.
- One down due to failure.
- This is called N+2 redendancy

Technique #2

Google “Canary” upgrade process

Canary (def'n)

- In the 1800s coalminers took canaries into the mines with them.
- Canaries are sensitive to toxic fumes.
- If the canary died, the miners would leave the mine.

"Canary in a Coal Mine" by The Police

First to fall over when the atmosphere is less than perfect

Your sensibilities are shaken by the slightest defect

You live you life like a canary in a coal mine

You get so dizzy even walking in a straight line

The Canary Process (overview)

- Used when upgrading 100s or 1000s of replicas.
- Basic premise:
  - Upgrade 1 machine.
  - Let it run.
  - If it doesn't fail after xx minutes, upgrade the remaining.

The Canary Process (detail)

- Remove one machine from LB.
- Upgrade it. Send tests. Add to load balancer.
- Wait 10 minutes.
  - Upgrade one per minute until 1% of all machines are done.
  - Upgrade one per second until all machines are done.

If there is a failure?

- Stop the process.
- Capacity is reduced but service is still provided.
- Revert to old version if capacity is needed.

Canary is not a testing process

- It happens **after** testing is done.
- You want tests to fail: (it finds problems)
- You don't want canaries to die: it means you didn't test enough.
- Canary is an insurance policy against insufficient testing.

If you use canary as a testing process, you are testing on live users. This is bad system administration.

Anecdote:

The canary that accidentally became a "testing on live users"

Technique #3

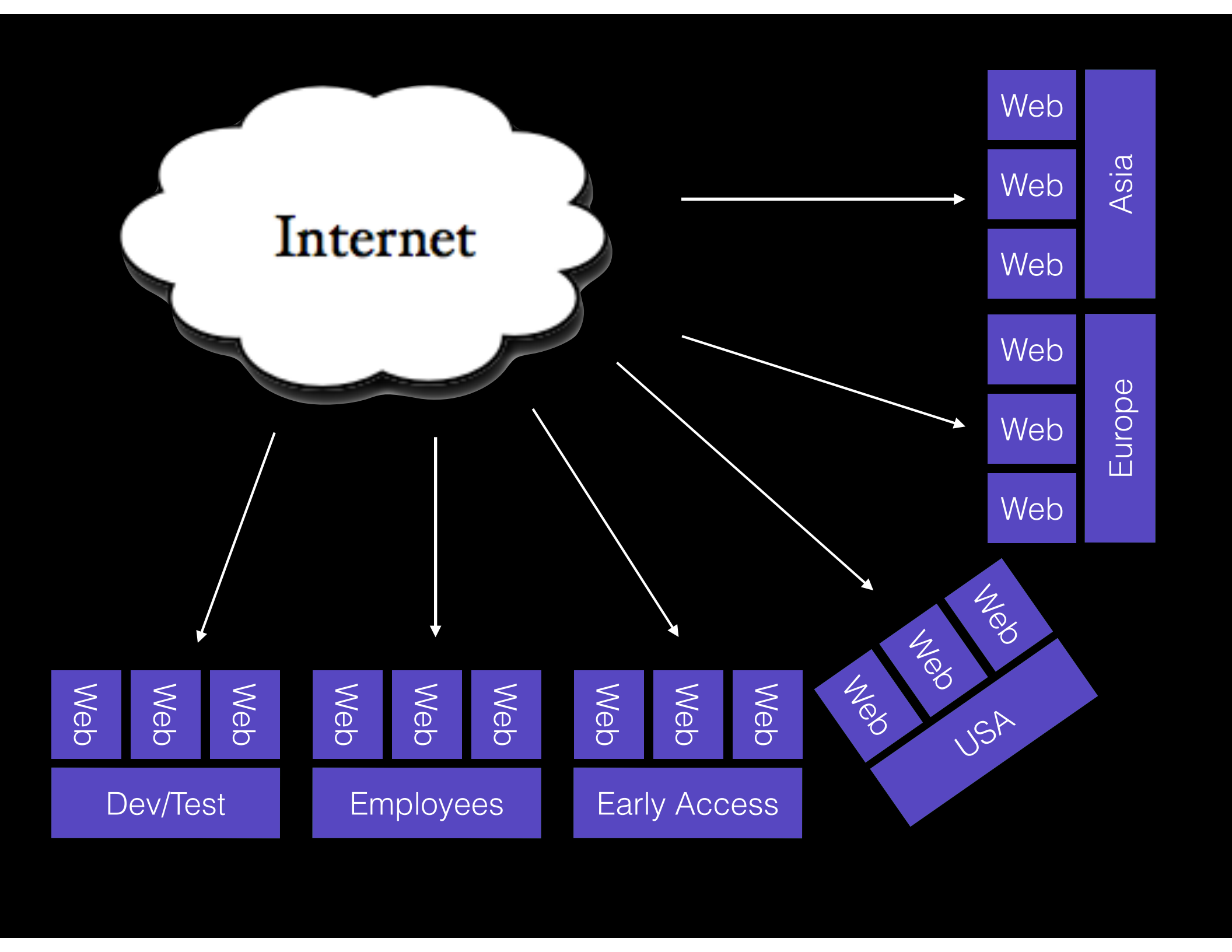
Phased Roll-outs

Phased Roll-outs

- Upgrade one group of users at a time

Facebook:

- Service is provided on many self-contained "clusters".
- Clusters are upgraded one at a time.
- employee-only cluster.
- beta-user cluster.
- global regions
- etc.


Internet
Web
Web
Web
Asia
Web
Web
Web
Europe
Web
Web
Web
USA
Web
Web
Web
Dev/Test
Web
Web
Web
Employees
Web
Web
Web
Early Access

- Use the same roll-out process in all environments.
- Automate to assure this.
- You are testing the roll-out process as much as you are testing the code.

Stack Exchange: Q&A software

130 web communities, plus meta's. (same software, different CSS)

StackOverflow.com
meta.StackOverflow.com

ServerFault.com
meta.ServerFault.com

parenting.stackexchange.com
meta.parenting.stackexchange.com

pets.stackexchange.com
meta.pets.stackexchange.com

meta.StackExchange.com

Phased Roll-Outs

- Dev environment
- Test environment
- Meta sites
- Less populated communities
- More populated communities
- StackOverflow.com itself

Healthcare.gov: 50 states on day 1

- What if they had started with Rhode Island?

Technique #4

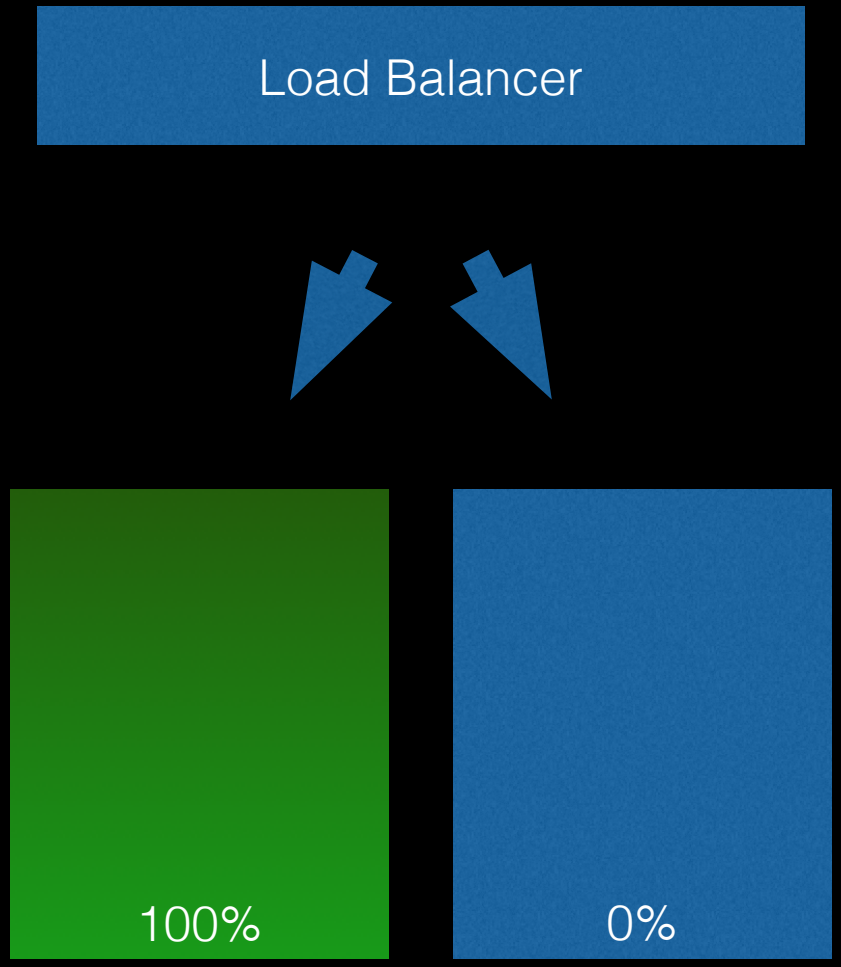
Proportional Shedding

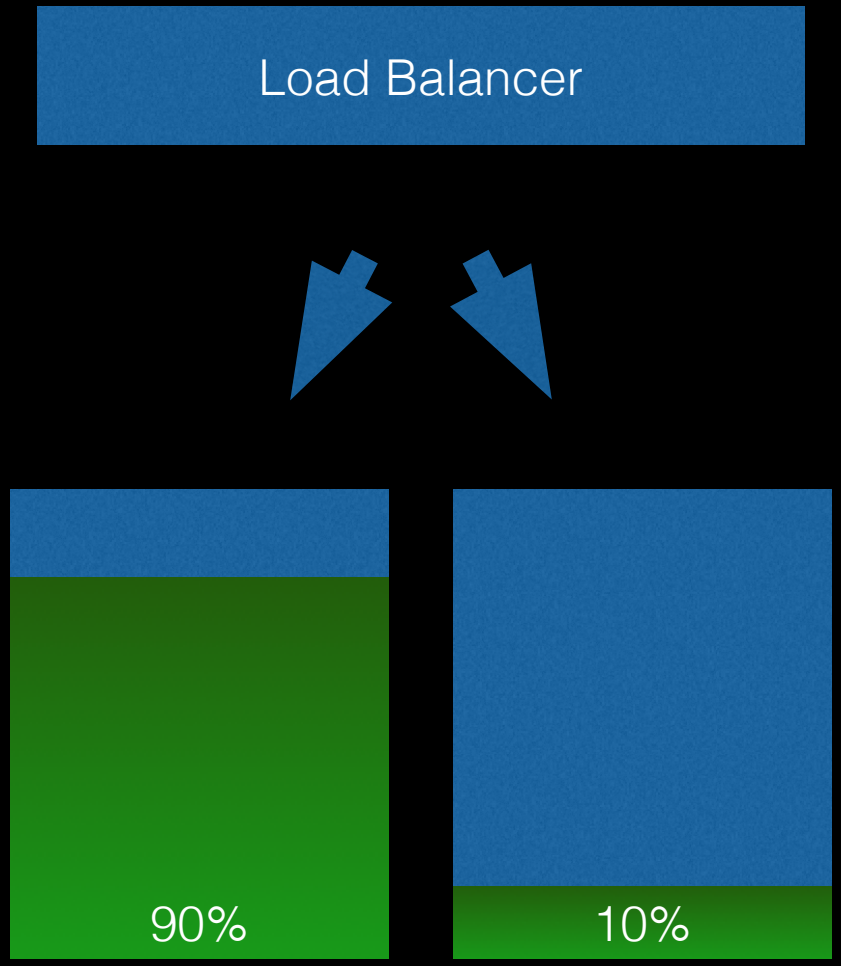
Proportional Shedding

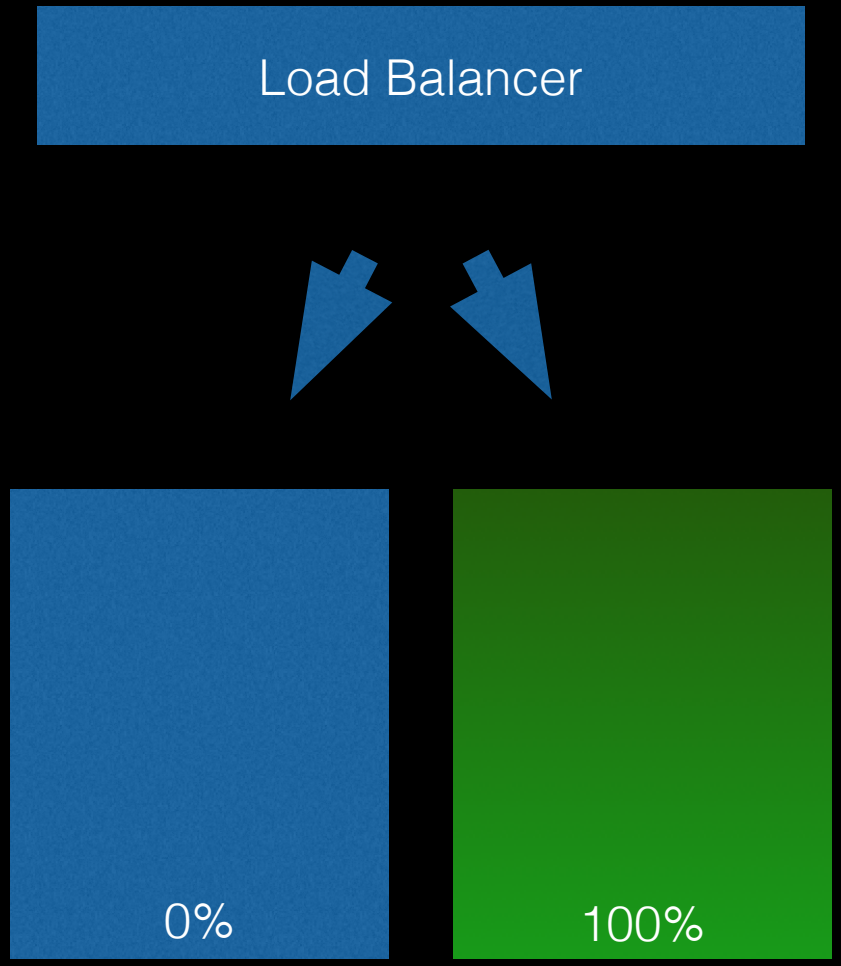
- Transition from old release to new release slowly over time.
- i.e. "Shed" the load.

Example:

- Cluster A: Old software
- Cluster B: Built with new software
- Load balancer sends 100% traffic to A
- Then shifts 1% traffic to B
- Then shifts 2% traffic to B
- ..
- Then shifts 90% traffic to B
- ...
- Then shifts 100% traffic to B


Load Balancer
100%
0%


Load Balancer
90%
10%


Load Balancer
0%
100%

Positives:

- Full capacity always available
- Change as fast or as slow as you wish.
  - 1% then wait (like a Canary)
  - 0 to 100% over an hour
  - 0 to 100% over the course of a day

Negatives:

- Requires 2x the capacity.

Proportional Shedding on 1 machine?

- Yes!
- If the service is self-contained in a subdirectory.
- Incoming traffic goes to Nginx or other load balancer.

Technique #5

Feature Toggles

Feature Toggles

- Tie new features to command-line flags.
- Start with flag=False.
- Set flag=True to enable feature
- Problems? Change flag=False.

Example:

- Suppose your site doesn't have a real-time spell-checker.
- Code is "hidden behind this flag".
- Code can be developed in stages, with feature off.
- Code can stay hidden for months.
- No "big merge" (reduces "bad merge" risk).

Command line flags

```
$ my-server --sp-algorithm-v2 \
    --morphological=off
```

Environment variables:

```
$ export SP_ALGORITHM_V2=yes
$ export SP_MORPHOLOGICAL=off
$ my-server
```

Flag files

```
$ cat spell.flags
sp-algorithm-v2=on
morphological=off
$ my-server --flagfile=spell.flags
```

Flag "lockfiles" (ZooKeeper)

```
my-server --zkflags=/zookeeper/config/spell
```

Uses:

- Rapid development
- Gradual introduction of new features
- Finely timed release dates
- Dynamic rollbacks
- Bug Isolation
- A-B testing
- 1 percent testing
- Differentiated services

Technique #6

Facebook's "Dark Launch" system

How do you launch a feature that will have 70 million users on day 1?

What could possibly go wrong?

- Inaccurate capacity estimates
- Unforeseen issues that only appear at big scale
- Bugs

Load testing 10,000 users will not reveal problems seen at 1 million users or 70 million users.

Simulate 10,000 users and multiple by 7,000. (won't work)

Small companies have it easy

- Starting from zero users, it is easy to grow slowly over time.

A big problem for big companies

- Facebook, Google, and others have to go from 0 to millions of users on the first try or it is headline news.

Hide behind a flag?

- Permits us to disable the feature.
- What if the "big announcement" is on Tue, December 2 and we want the feature to go live at the same time?

Soft launch:

- Feature is enabled but not announced.
- Usage grows slowly due to "word of mouth".
- Feature can be disabled entirely because it isn't "official" yet.

Dark Launch for Facebook Chat

- Would have 70 million users on the first day.
- Could not accurately predict:
  - capacity needed on "day 1".
  - problems at this scale.
- Systems have become too big and too complex to be predictable.

Dark Launch:

- Feature is enabled but invisible
- Feature sends artificially generated data to chat system
- Capacity is tested
- Issues found early

How to select users?

- A random 10,000 users?
- 1% of all users?
- (Growing to all users over time)

Facebook Gatekeeper

- "Pushing Millions of Lines of Code Five Days a Week" (Facebook, 2011, Chuck Rossi)
- "Gatekeeper" permits fine-grained control over which features are revealed to which users.
  - Country
  - Age
  - Datacenter
  - Is user a known TechCrunch employee

The code for every feature Facebook will announce in the next six months is probably in your browser already.

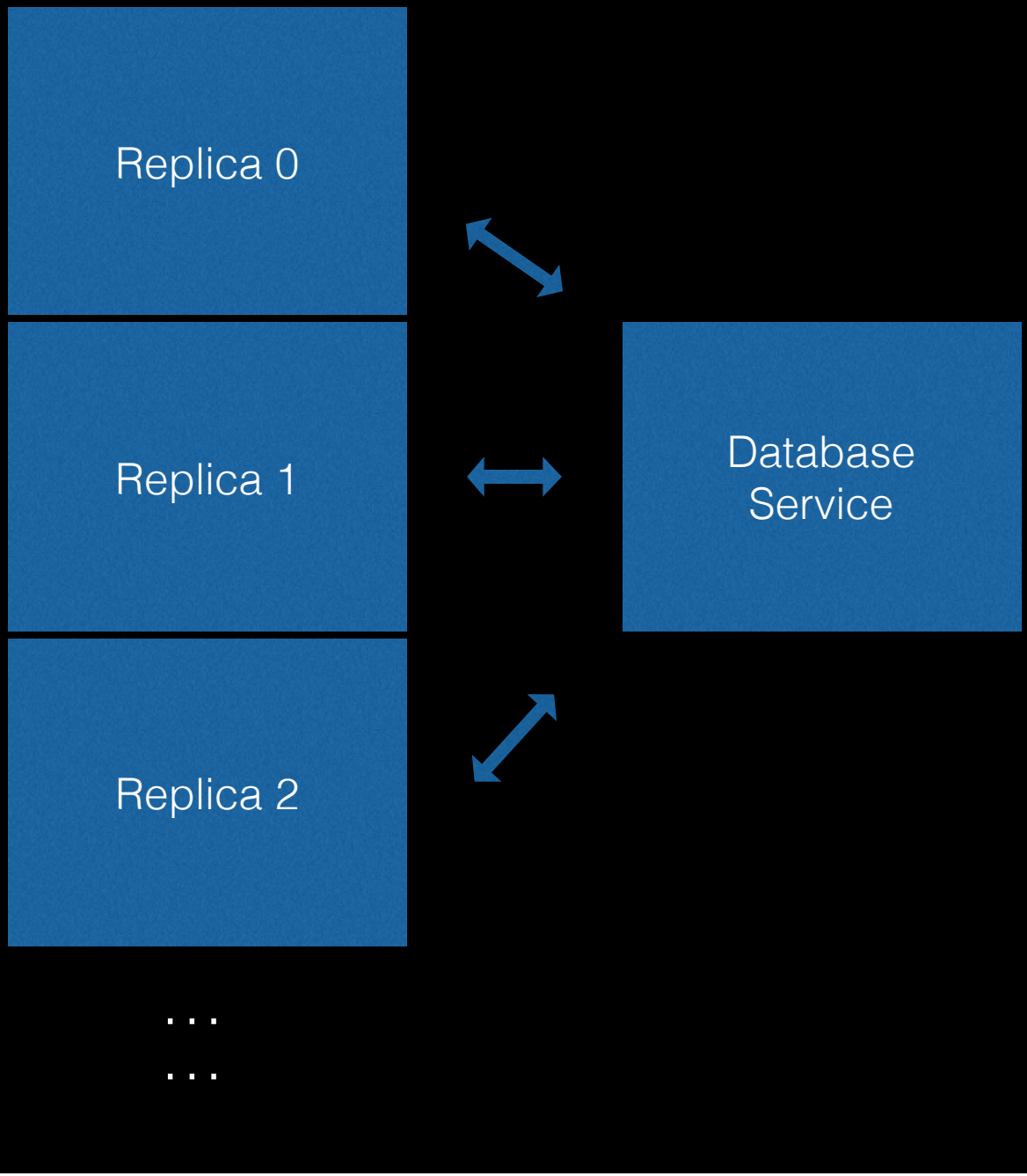
Scientific discoveries often come from testing things that don't need to be tested, and getting unexpected results.

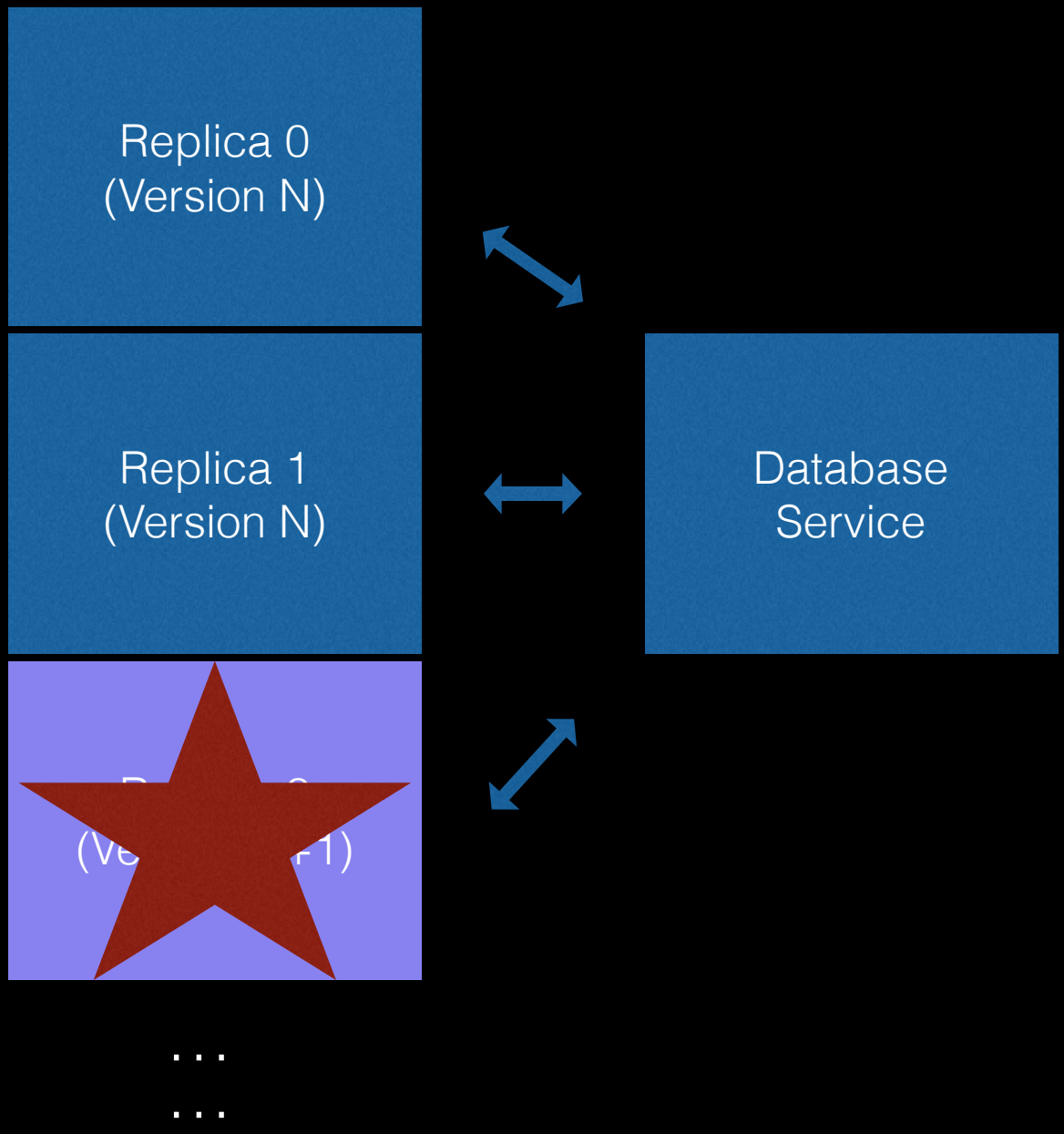
Google's IPv6 Dark Launch

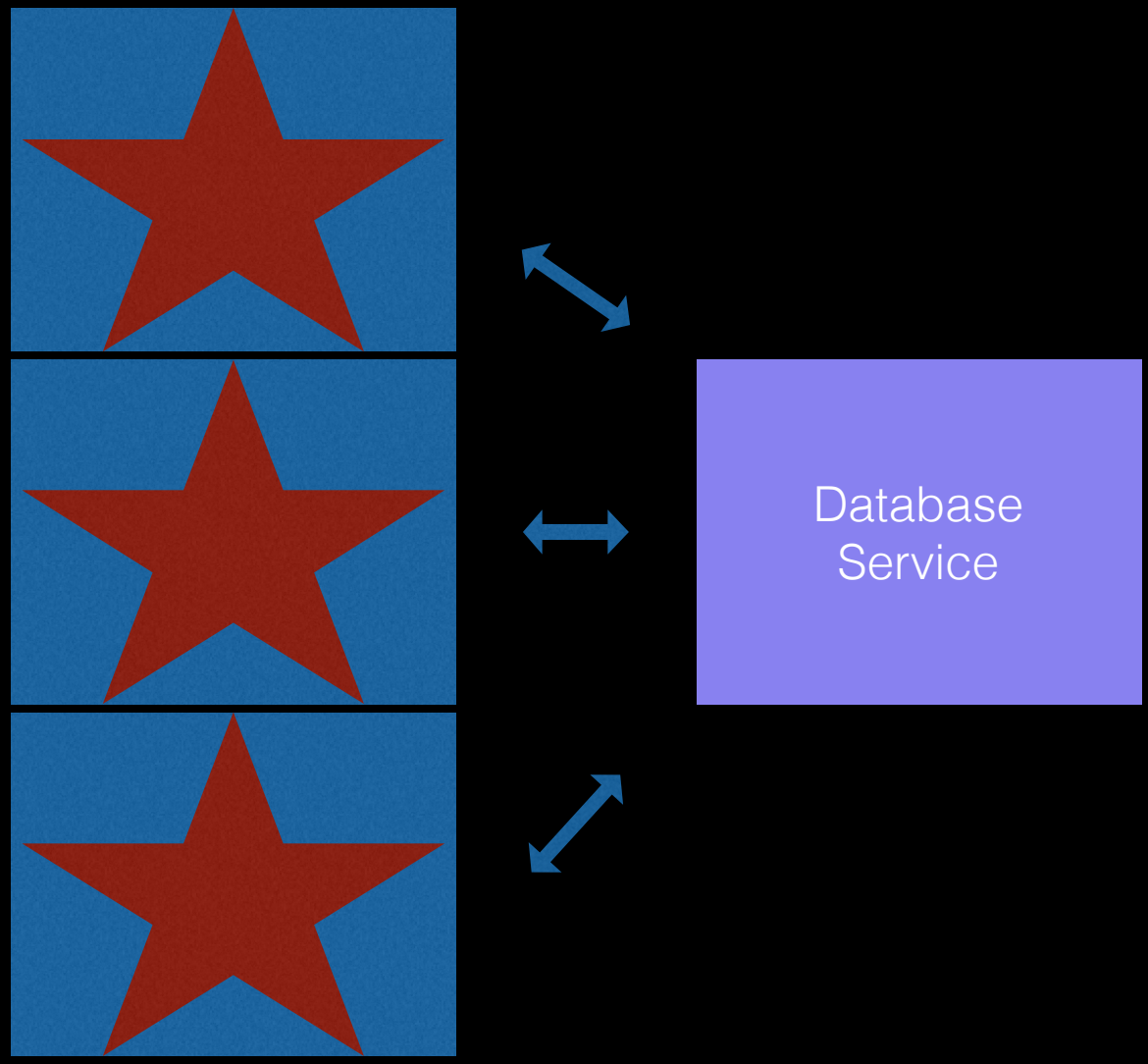
- Injected JavasScript code that queried for an image available via IPv4 and IPv6.
- "Shouldn't have any problems"
- Found operating system bugs, IPv6 "islands".
- Quantified Risk.
- Enabled fixing problems before visible.
- Reduced risk of highly visible "black eye" for IPv6.

Technique #7

Live Schema Changes


Replica 0
Replica 1
Replica 2
Database Service
...
...


Replica 0
(Version N)
Replica 1
(Version N)
Database
Service
. . .
. . .


Database
Service
. . .
. . .

Why is this difficult?

- Catch-22: Can't change the database first. Can't change the code first.
- With many of replicas, can't upgrade them all at once.
- Can't canary.

If you only add new fields, you are ok.
(But what about complex changes?)

SQL Views

- Use "Views" to hide the changes.
- Programmers code to the "view" API.
- Change the schema? Change the view code at the same time.

Example:

- Before schema change: View accepts new fields and semantics, translates.
- After schema change: View is a "no op".

What about systems with no "views"?

- Views are a PITA because they require discipline.
- Some systems don't support views.

The McHenry Technique

- 5 phases with clearly defined back-out steps for each one.

Overview:

- Replicas upgraded to handle old and new schema.
- Schema is changed.
- Replicas upgraded to handle only new schema.

Phase 1:

Original software, Original Schema

- The running code reads and writes the old schema, selecting just the fields that it needs from the table or view.
- This is the original state.

Phase 2: "Expand"

Original software, **Schema + New Fields**

- The schema is modified by adding any new fields, but not removing any old ones.
- No code changes are made.
- If rollback is needed, it's painless because the new fields are not being used.

Phase 3:

Dual-Compatible software, Schema + New Fields

- Code is modified to use the new schema fields and pushed into production.

If rollback is needed, it just reverts to to phase 2.

At this time any
data conversion
can be done while the
system is live.

Phase 4: "Contract"

New "Pure" Software, Schema + New Fields

- Code that references the old, now unused, fields is removed and pushed into production.
- If rollback is needed, it just reverts to phase 3.

Phase 5:

New “Pure” Software, **New “Pure” Schema**

- Old, now unused, fields are removed from the schema.
- In the unlikely event that a rollback is needed at this point, the database would simply revert to phase 4.

Optimizations:

- Combine or overlap Phases 4 & 5.
- Lazily remove old fields in future releases.
- Phase 5 from schema version n is combined with Phase 2 from schema version n+1.

Technique #8

Languages support for live code upgrades

Languages support for live code upgrades

- Geordi "rewrote the subroutine" live, right?
- Generally frowned upon in real life, but what if language supports it?

Erlang

- Properly structured Erlang programs are designed as event-driven finite-state machines (FSM).

The Details

- For each event received, a specific function is called.
- One "event" is the "code has been upgraded" event.
- Function assigned to that event upgrades any data structures in-place.
- All subsequent events trigger functions from the new software version.
- This requires careful planning and procedures. See:
- http://learnyousomeerlang.com/what-is-otp

	Phone Off	Phone Dialing	Call Connecting	Call In Progress	Call Disconnecting
Key Pressed	Func AA	Func AB	Func AC	Func AD	Func AE
Audio Received	Func BA	Func BB	Func BC	Func BD	Func BE
Audio Sent	Func CA	Func CB	Func CC	Func CD	Func CE
UPGRADE	Func U	Func U	Func U	Func U	Func U

Bringing it all together

Continuous Integration:

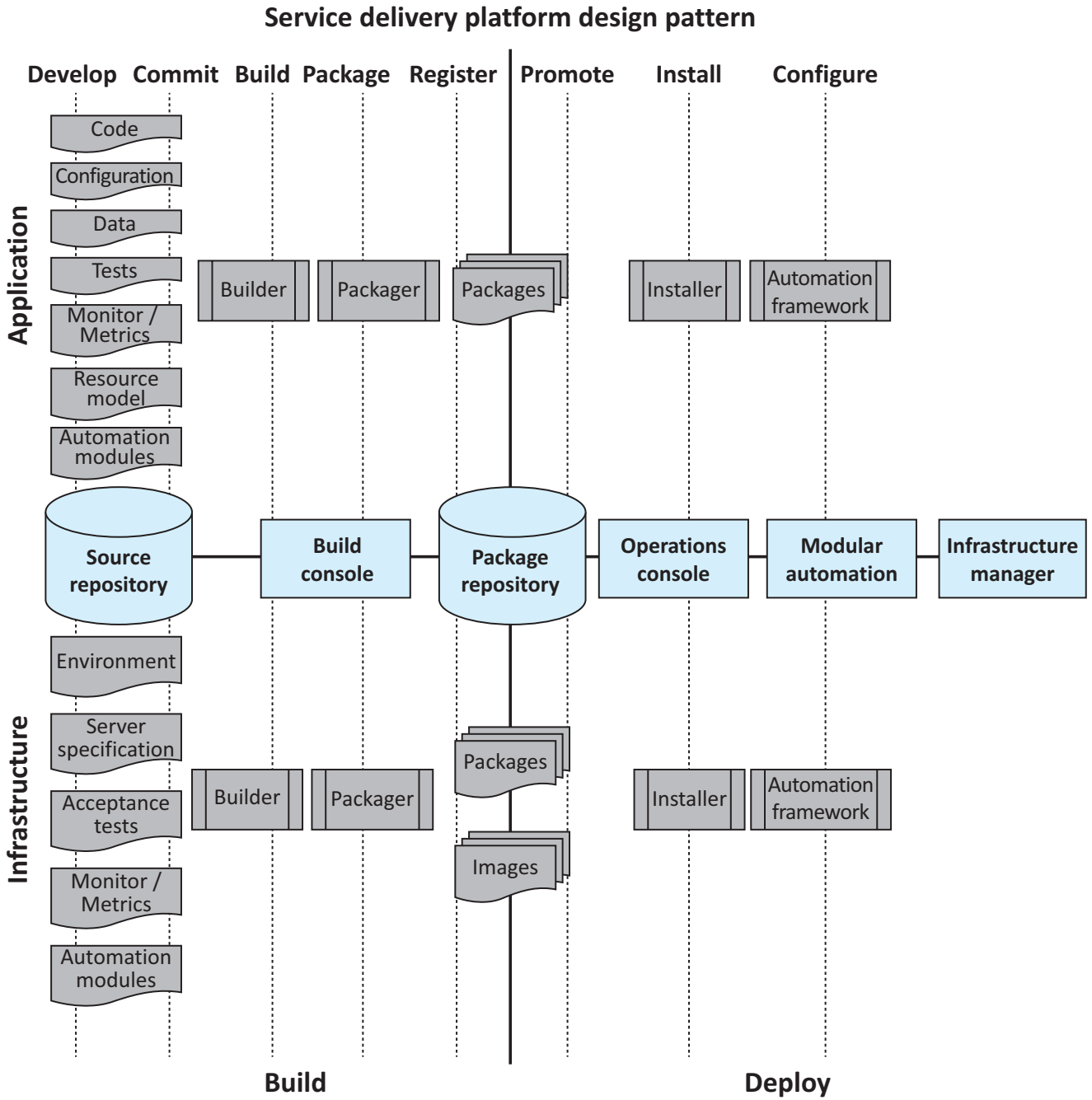
- Every code change results in compile + test

Continuous Delivery:

- Every CI results in automated testing; if passed the packages are fit for deployment.

Continuous Deployment:

- Every successful CDelivery results in deployment to an environment (test, beta, prod)



Google Example:

- Making "Push On Green" a Reality: Issues & Actions Involved in Maintaining a Production Service
- Usenix LISA 2014

Techniques discussed:

0. Take the system down for upgrade
1. Rolling upgrades
2. Google's "canary" upgrade system
3. Proportional Shedding
4. Feature Toggles
5. Facebook's Dark Launch system
6. Upgrades that involve schema changes.
7. Languages that support live code upgrades
8. Continuous Deployment

Summary:

- Systems should be re-engineered to be upgraded live.
- There are many techniques, and many variations.
- Once automated, upgrades can be more frequent, with higher confidence.

Ability to Upgrade is becoming a business imperative

- It accelerates the ability to roll out new features and improvements.
- It is a prerequisite to maintaining security. (ShellShock, anyone?)

Live Upgrades on Running Systems:

8 Ways to Upgrade a Running Service With Zero Downtime

Tom Limoncelli @YesThatTom
EverythingSysadmin.com

With material from…