Managing Deployments in the Age of the Microservice

samschaevitz at google dot com
Gmail & Calendar SRE @ Google

Prepared for SREcon EMEA 2017
Dublin, Ireland
2017-08-31

Agenda

1. Types of Service Changes
2. Why Change Things At All?
3. What Can Go Wrong?
4. Best Practices
5. A Brief Interlude on Naming
6. Payoff

Caveat:
This talk is targeted for those who work on applications composed of layered, replicated, sharded, lightweight and fine-grained services.

Types of Service Changes

Control Surfaces

Type	**Time to Recovery**
Client Change	O(Day)
Server Binary Change	O(Hour)
Static Configuration Change	O(Hour)
Dynamic Configuration Change	O(Minute)

Why Change Things At All?

This is reliability engineering.
The safest thing to do is to
never change anything!

“A ship in harbor is safe,
but that’s not what ships are built for.”

- *John A. Shedd*

The ~~safest~~ **most optimistic and least rewarding** thing to do is to never change anything.

Is a Change Freeze Right For You?

- ➔ your feature set is complete, forever and ever
- ➔ your system never ever breaks, and will never break
- ➔ even if it did, you totally already understand every single failure mode of your system
- ➔ your infrastructure or external dependencies never change
- ➔ you cannot possibly optimize your costs better than you already have
- ➔ wouldn't ever want to adopt any new technologies to make your infrastructure more reliable than it already is

“You never know what’s hiding inside these systems. You never know exactly what’s going to trigger it.”

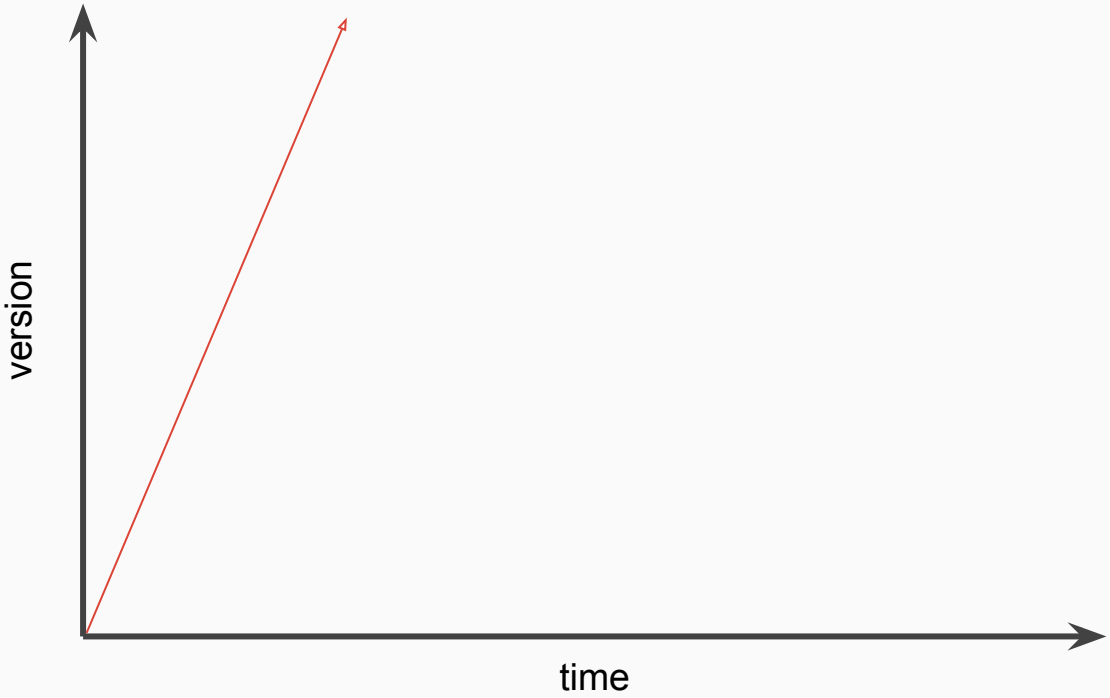
- *Bill Curtis, SVP & Chief Scientist at CAST*

“Following the major IT system failure experienced earlier today, with regret we have had to cancel all flights leaving from Heathrow and Gatwick for the rest of Saturday,” a spokeswoman said.

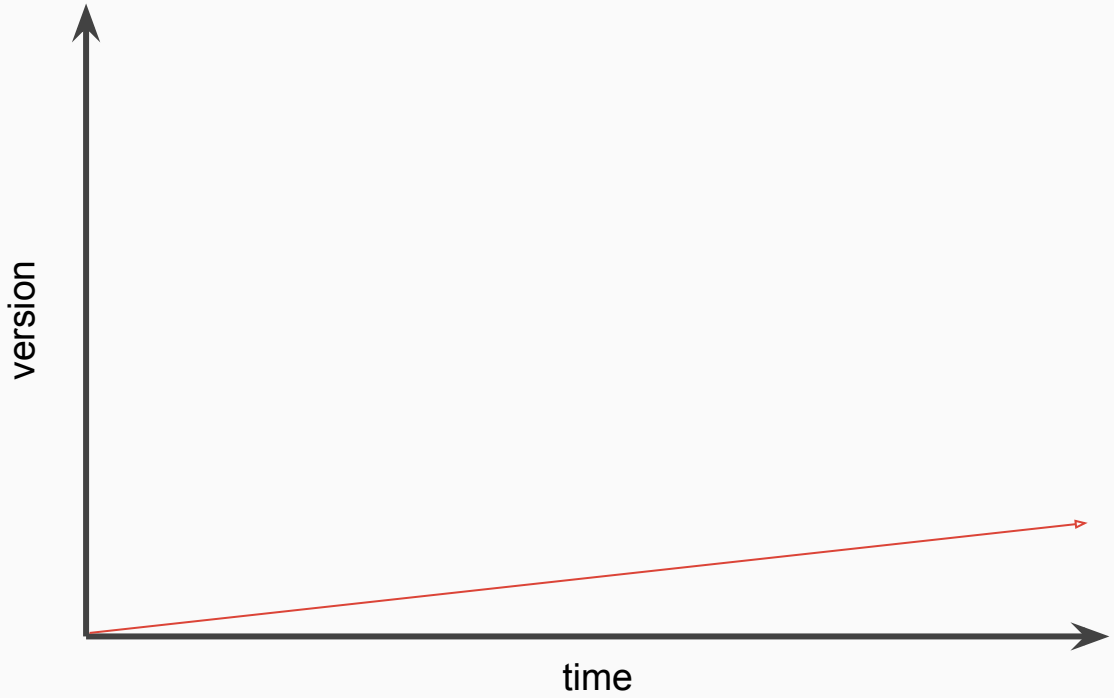
“The Delta Air Lines technology glitch that canceled hundreds of flights Sunday and Monday was smaller than other episodes in recent months that cost airlines tens of millions of dollars, but it still served as a reminder of how fragile airline computers are.”

image: pexels, cc0

“A computer problem forced United Airlines to ground all domestic flights for about an hour on Sunday evening, causing a cascade of delays and annoying customers throughout the United States.”



Ideal Change Velocity, the Dev organization’s perspective



Ideal Change Velocity, an SRE’s perspective

How do we manage these conflicting incentives?

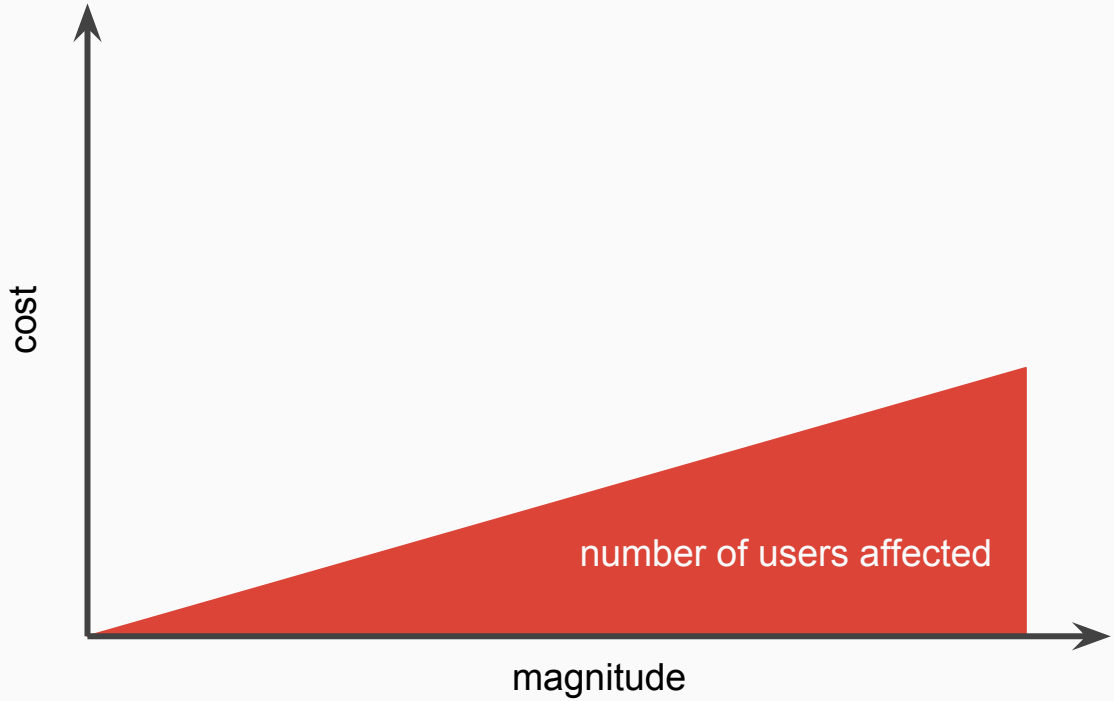
Align increases in risk exposure with decreases in risk profile.

What Could Go Wrong?

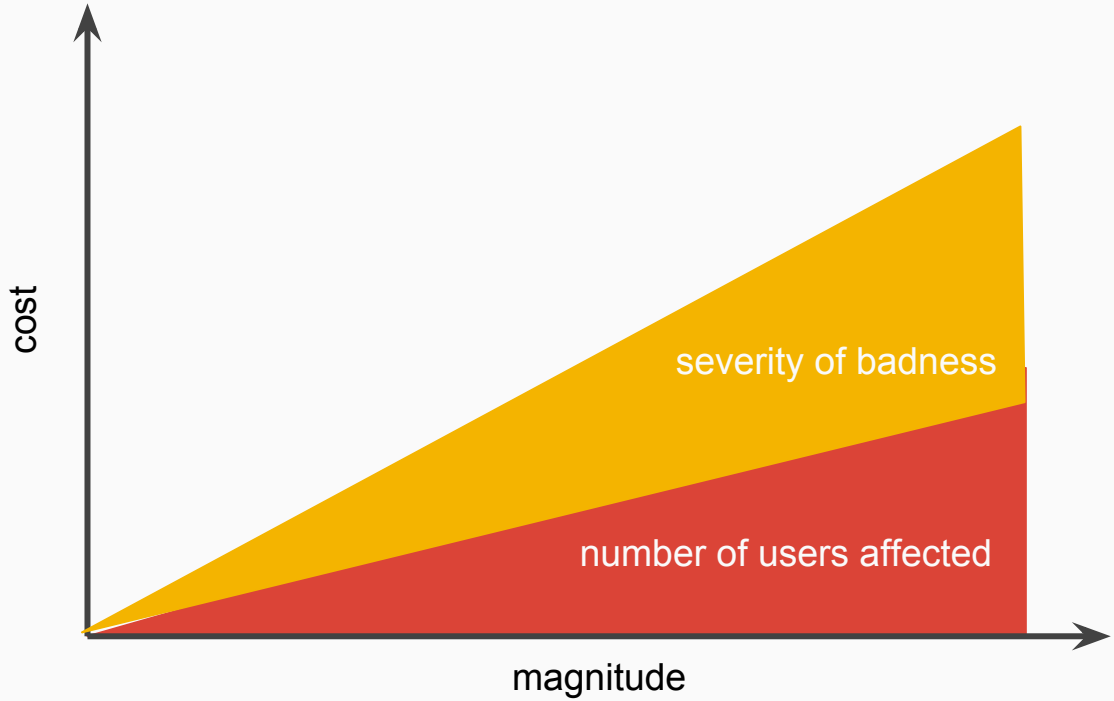
Examples of System Badness

- ➔ the client doesn’t show the user the error and silently/automatically retries and succeeds
- ➔ end user-visible latency when you load your Gmail inbox increases by 10% at the 99th percentile
- ➔ a common Java library has been updated to use a new implementation of HashMap and it costs 10% more CPU cycles for our main application service to run
- ➔ Google Wallet integration is broken in the compose box of Gmail
- ➔ redesigning the UI layout and users hate it
- ➔ every user trying to access mail.google.com receives a Server 500 error

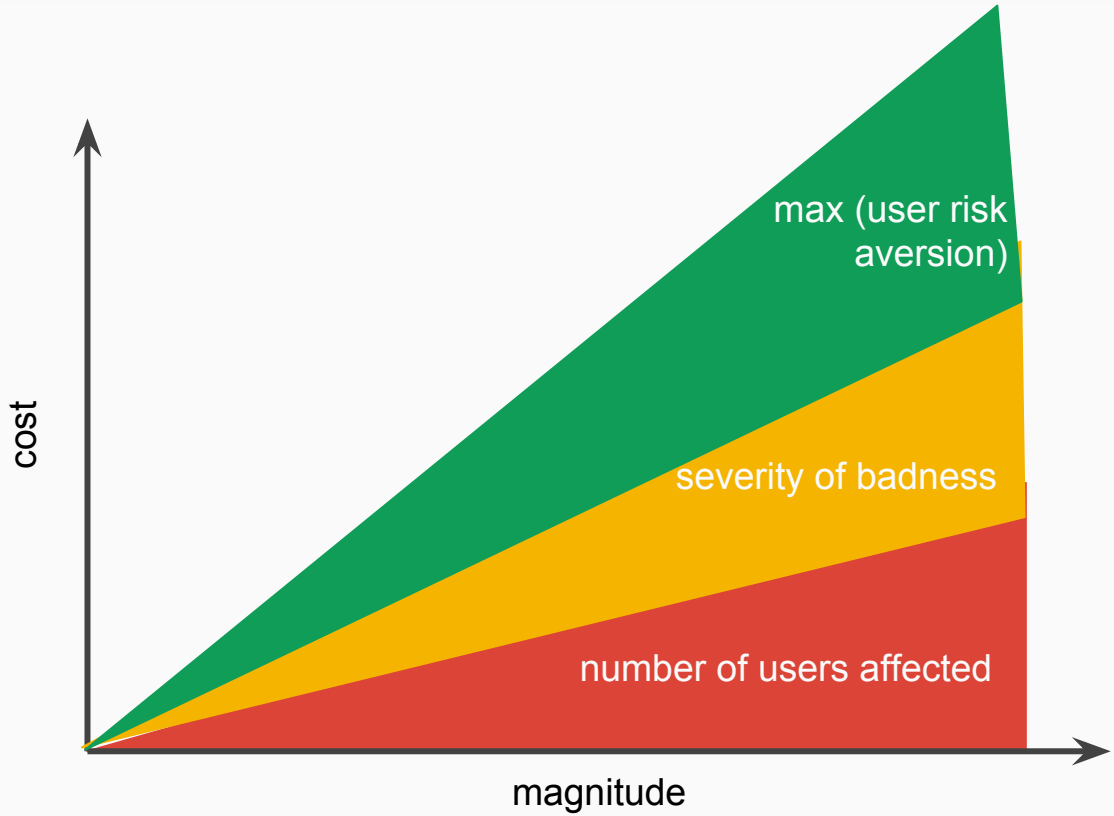
Cost of Badness



Cost of Badness



Cost of Badness



Calculating Cost of System Badness

$$\text{Total Disruption Cost} \approx (\text{Number of Affected Users} * \text{Disruption Severity})^{-\text{Max. User Risk Aversion}}$$

Penalty Buckets

- ➔ immediate lost revenue
- ➔ user trust
- ➔ contract violations
- ➔ increased infrastructure costs
- ➔ engineering time

User Types

- ➔ developer team testers
- ➔ internal testers
- ➔ trusted external testers
- ➔ free-tier consumers
- ➔ paying customers

Best Practices

The Optimal Rollout Is...

- ➔ staged
- ➔ progressive
- ➔ revertable
- ➔ transparent
- ➔ automatic
- ➔ well-understood

Developer Team Testing

- Buildable
- Runnable
- Obvious feature breakages

Internal Testing

- More subtle feature breakages
- Obvious performance regressions

Trusted External Testing

- Do our paying customers (with higher risk tolerances) notice anything they don’t like about a new release?

Some Free-Tier Users

- Nuanced performance regressions
- Subtle bugs

Remaining Free-Tier Users

- Subtler performance regressions
- Subtler feature regressions

Paying Users

- Time in earlier stages is enough to deploy to most risk averse, most expensive to lose customers

To Detect Badness as Early As Possible...

- Have high-quality (unit, integration) test coverage
- Build your targets as often as possible
- A/B experiment everything
- Automate to limit toil, risk of human error
- Hold back some vocal internal users for the duration of the launch
- Deploy often; limit change surface
- Make it easy to rollback
- Encourage backwards compatibility in code

A Brief Interlude on Naming

Less Important: Actual Names

More Important: Consistent Names

More Important: Consistent Names

- ➔ server name schemas
- ➔ binary release stages
- ➔ machines
- ➔ load-balancing domains
- ➔ user populations

Payoff

In Return for A Principley-Engineered Deployment System...

- ➔ Detect problems long before the exposure costs you time or money
- ➔ More, better automation systems
- ➔ Easier debugging
- ➔ Automatic rollbacks of bad deployments
- ➔ Frequent production deployments
- ➔ Happier, more productive organizations

A developer is happy because her code has launched.

Questions?

Thanks!

Extras

Glossary

binary push - installing a different binary version of a long-lived server, starting it, and migrating traffic previously handled by the old version of the process to the new version.

client update - releasing a new version of web, native or mobile application code to users, giving them the option to update their native or web applications.

static configuration change - a change that requires restarting the same version of your application with e.g. new command-line flags read a startup time.

runtime configuration change - a change to a long-lived server that does not require a restart to take effect e.g. a list of healthy backend services that are able to receive traffic from this service.

feature launch - the enabling of new functionality in your application, such that it is usable by your customers.

dark launch - the exercise of new code in your application without affecting the user's experience of your application in order to gain confidence in new features before the cost of rolling back is too high.