Reliable Software Systems

Week 5: Production
Windows 10 October 2018 update

“Windows as a Service” ships fixes and feature updates to customers’ PCs.

Early testers noted that this update deleted files from certain directories.

It rolled out anyway, but was halted after more customers reported data loss.

After the fix, they rolled it out more slowly, watched metrics.

Incompatibilities with drivers & some apps caused more delays as some users were restricted from updating.

Eventually (over two months later) all restrictions lifted.

Brenan Keller
@brenankeller


First real customer walks in and asks where the bathroom is. The bar bursts into flames, killing everyone.

1:21 PM - 30 Nov 2018

24,785 Retweets 61,892 Likes

im bird @SomeEgrets · 1 Dec 2018

Replying to @brenankeller @ElenaKotka
Now, now. To be fair, there was no such thing as a bathroom on the original hardware. They were supported in Bar 4.0, fifteen years ago, and most people are running Tavern, Pub, or Speakeasy these days anyway.

But our critical ops are tightly coupled to Bar 1.0, so we're stuck.

1 Retweet 15 Likes
# Google’s top 8 outage triggers

<table>
<thead>
<tr>
<th>Trigger</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary push</td>
<td>37%</td>
</tr>
<tr>
<td>Configuration push</td>
<td>31%</td>
</tr>
<tr>
<td>User behavior change</td>
<td>9%</td>
</tr>
<tr>
<td>Processing pipeline</td>
<td>6%</td>
</tr>
<tr>
<td>Service provider change</td>
<td>5%</td>
</tr>
<tr>
<td>Performance decay</td>
<td>5%</td>
</tr>
<tr>
<td>Capacity management</td>
<td>5%</td>
</tr>
<tr>
<td>Hardware</td>
<td>2%</td>
</tr>
</tbody>
</table>

From *The Site Reliability Workbook*, copyright Google Inc, used under CC BY-NC-ND 4.0
Deployment models

- Less frequent releases
  - More structured releases
    - Scheduled full-product release
  - Release-at-will
- More structured releases
  - Scheduled team-based release
  - Push-on-green
  - Continuous Deployment

- Less structured releases
  - More frequent releases
Mitigating risk

**Blue/Green deployments**

- Have two separate production environments ("blue" and "green")
- Deploy to blue
- Move traffic from green to blue
- If problems, move the traffic back to green
Mitigating risk

Canarying

Deploy new code to a small part of your production environment
Send a small amount of traffic to the canary
Monitor to make sure the canary is healthy
If problems, move traffic back to the rest
Otherwise, continue rolling out
Mitigating risk

Gradual release

Deploy to a small part of production

Send part of your users there

If good, gradually increase the rollout and the amount of traffic
Mitigating risk

Feature toggles - configuration that turns features on or off

```java
if flag("use_my_super_cool_new_feature") {
    // do my new awesome stuff
} else {
    // do the boring old stuff that works
}
```
Multiple versions of code running

Clients

Servers

Dependency

Database

Dependency
Multiple versions of code running

Example: to move a field from one database to another

1. Code double writes - write to both the old database and the new.
2. Release.
3. Migrate all old data from the old database to the new format.
4. Code changes to read from the new location.
5. Release.
6. Code removes old write and read path.
Oh no, I broke it!

If still have the old deployment around...

   Redirect users to the stable deployment

If there’s a feature toggle...

   Turn it off!

If rollout is finished...

   Prefer rolling back rather than fixing the code
Assorted Best Practices

Automate repeated processes

Don’t release on Fridays :)

Consider a “production freeze”

Make sure there’s an audit trail

What was released when?

Show build versions in your monitoring!
Testing in production

I DON'T ALWAYS TEST MY CODE

BUT WHEN I DO, I TEST IT IN PRODUCTION
Testing in production

There will be failures anyway.

So practice failing* and recovering**!

* requires buy-in from your company

** and careful planning and engineering
Disaster recovery testing

Google annually practices failure by breaking live systems and testing procedures

**Large**: take down a data center

**Small**: team takes down a few servers

**Real**: turn off some internal tools

**Procedural**: have datacenter operators pretend to buy massive amounts of diesel fuel
Chaos engineering

Started by Netflix in 2010 to vet their move to AWS

“Chaos Monkey” would pseudo-randomly reboot machines

Later added more types of failure injection

Latency, error rates...

Also added ways to scope failure and have a feedback loop
END