DEPLOYING OFTEN IS A VERY GOOD IDEA

by Dan McKinley

@mcfunley

Hey, I wrote these slides for a different talk and had to cut them out. But I think they're ok!

What are our goals?

What the hell are we trying to do in software?

Well, if we’re talking about an application on the internet, I can think of two big goals.

We have to ship stuff to the web

We have to ship a bunch of stuff. It’s our job as engineers to change things. That is the job.

We want to avoid breaking production

And, we don’t want to destroy everything of value that we’ve already built when we do that.

OBSERVATION

We can’t avoid breaking production.

The first thing you might notice about those goals is that the second one isn’t possible.

Or at least all industry experience suggests this. No significant web service yet conceived has never been down. S3 was down a while back. It was chaos.

We can only minimize the
time that production is broken.

The only thing we can do then, here on Earth, is try to minimize the length of time that production is broken.

How long is production broken?

`P(break | deploy)`

Anytime we deploy code, there’s some chance that it’s going to break something. We’ve got a probability of breakage, given a deploy.

How long is production broken?

`deploys • P(break | deploy)`

And that gets multiplied by the number of times we try to deploy code.

How long is production broken?

`deploys • P(break | deploy) • avg time to fix`

And then the time that production is down over some interval is the average time it takes us to fix it, times the number of times we take it down.

Dependent variables!!!

`deploys • P(break | deploy) • avg time to fix`

The key insight of continuous delivery is that these aren’t fixed quantities. In practice each of them actually depends on the values you pick for the others.

SOMEWHAT COUNTERINTUITIVELY,

deploys • **P(break | deploy)** • avg time to fix

Declines if you deploy frequently.

It turns out that if you crank up the number of deploys you do, the probability that any given deploy breaks declines.

SOMEWHAT COUNTERINTUITIVELY,

`deploys • P(break | deploy) • avg time to fix`

Declines if you deploy frequently.

So does this.

The time to fix a given broken deploy also declines if you crank up the number of deploys.

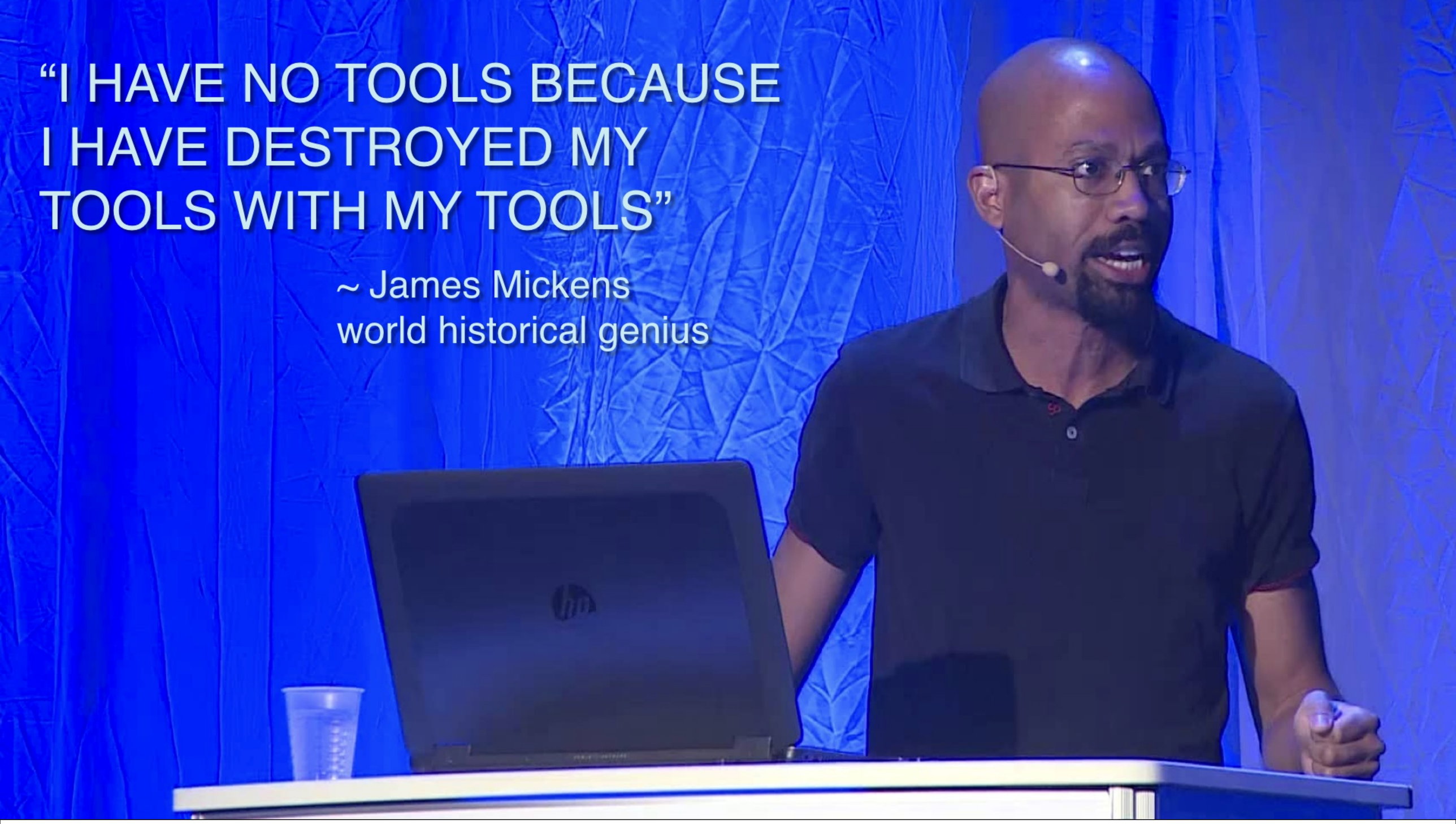
Deploys—the first term—obviously increase.

But I have become a believer that cranking it up is a win holistically, and has the practical effect of minimizing the value of

OBSERVATION

Risk increases with time between releases.

One reason this is true is that the longer it's been since we last deployed, the more likely it is that the next deploy is going to break.

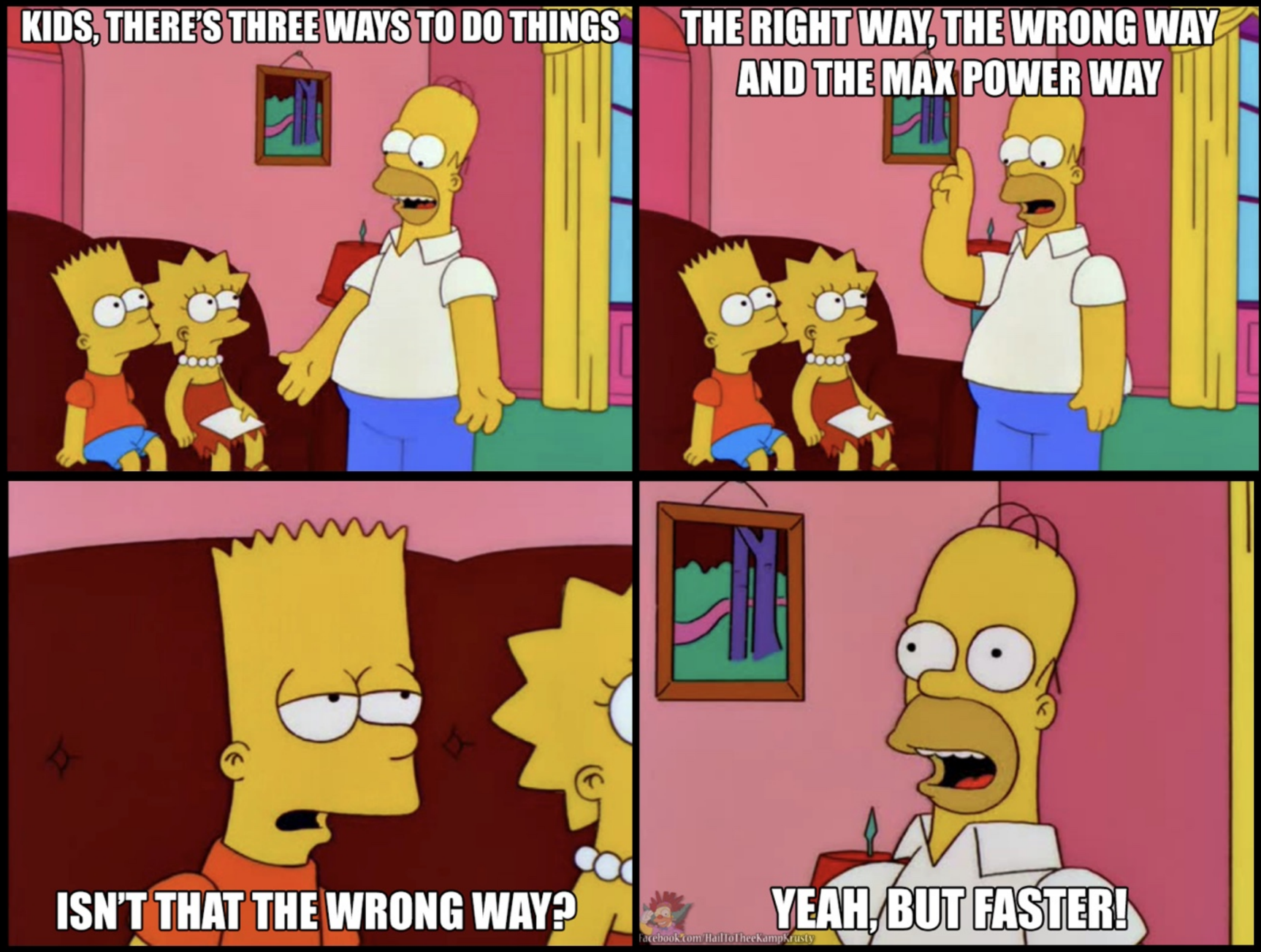


If you haven’t deployed since last month, the deploy tools themselves are most likely broken. You’ve either broken them directly with untested changes, or you’ve got IP addresses hardcoded in there and your infrastructure has changed, or who knows what.

OBSERVATION

Risk increases with slower deploys.

Another less-than-intuitive thing we should consider is that if deploys take a long time, this is dangerous. Even if they work reliably, slow deploys are not neutral.



If the deploy tooling isn't made fast, there's probably a faster and more dangerous way to do things and people will do that instead. They'll replace running docker containers by hand. They'll hand-edit files on the hosts.

So we want to make the "right way" to ship code also the laziest possible way.

OBSERVATION

We have to push on short notice even if we don’t want to.

Even if we could avoid shipping quickly most of the time, we’ll occasionally have to push in a big hurry.

Bad things happen in production, and they have to mitigated quickly. An actively-exploited SQL injection flaw is one example. These things come up in real life.

And when they do, you don’t want to be trying to use a poorly tested “fast path” in a crisis. That’s making an already bad situation downright dangerous.

OBSERVATION

Risk increases with time between coding and pushing.

If you deploy infrequently, that also means you wrote the code farther into the past. That’s bad.

203,847 commit results

Sort: **Best match**

what was I thinking?

majewsky committed to sapcc/limes 2 days ago

3e2ea86

What was I thinking...

SOF3 committed to poggit/poggit on **GitHub** 17 days ago

1c2127b

what i was thinking

thanapongp committed to thanapongp/attend-check-server 17 days ago

4e355e7

What Was I thinking

c9a3fee

You have great understanding of what code is doing when you're writing it. Then your comprehension of it gets strictly worse over time.

After a week or so you may barely have any idea what it was you were trying to accomplish. This is bad news if you find yourself having to debug a problem with it in production after deploying it.

OBSERVATION

Risk increases with the number of lines pushed.

And if you’re deploying infrequently, you’re also shipping a lot more code at once. This is a terrible idea.

This repository Search Pull requests Issues Gist

derp / derp

Code Issues 0 Pull requests 1 Projects 0 Wiki Pulse Graphs

my huge pull request #223

Open **mcfunley** wants to merge 63 commits into `derp:master` from `mcfunley:master`

Conversation 1 Commits 63 Files changed 126

Changes from **all commits** **126 files** +2,272 −467

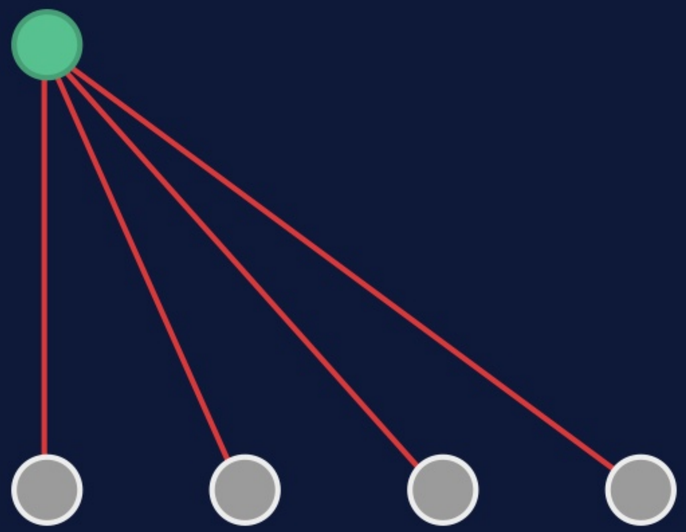
Every senior engineer knows to look at this page with a high level of panic. This is a merge that is definitely not going to go well. You can feel it in your bones.

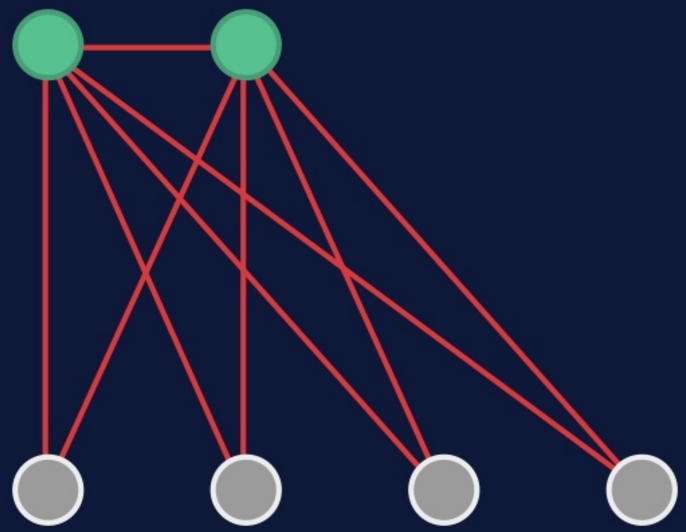
Every single line of code you deploy has some probability of breaking the site. So if you deploy a lot of lines of code at once, you’re going break the site. You just will.

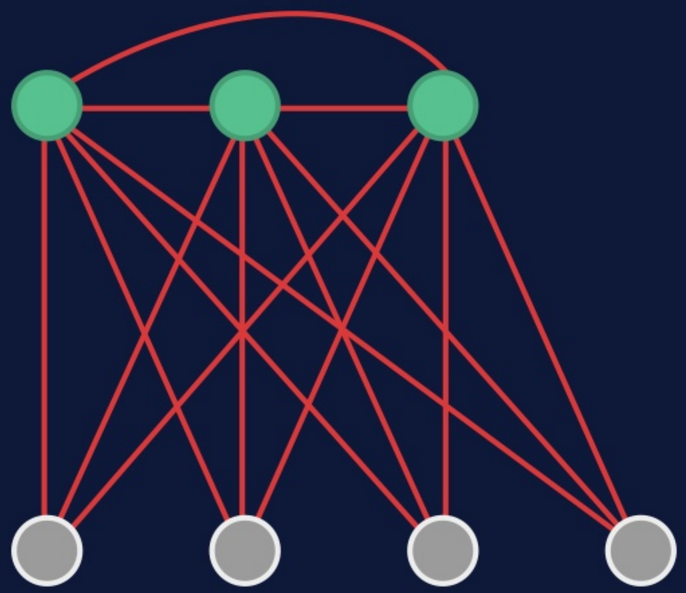
OBSERVATION

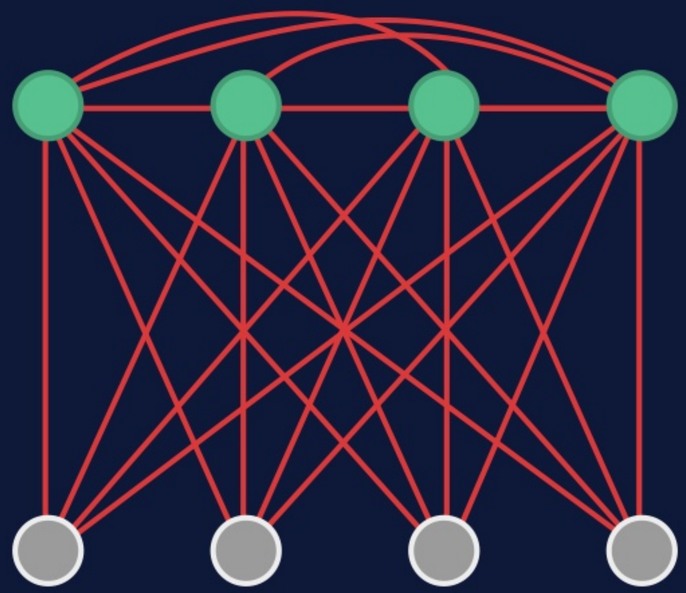
Debugging/review time increases with the number of lines pushed.

You also stand a better chance of inspecting code for correctness the less of it there is.



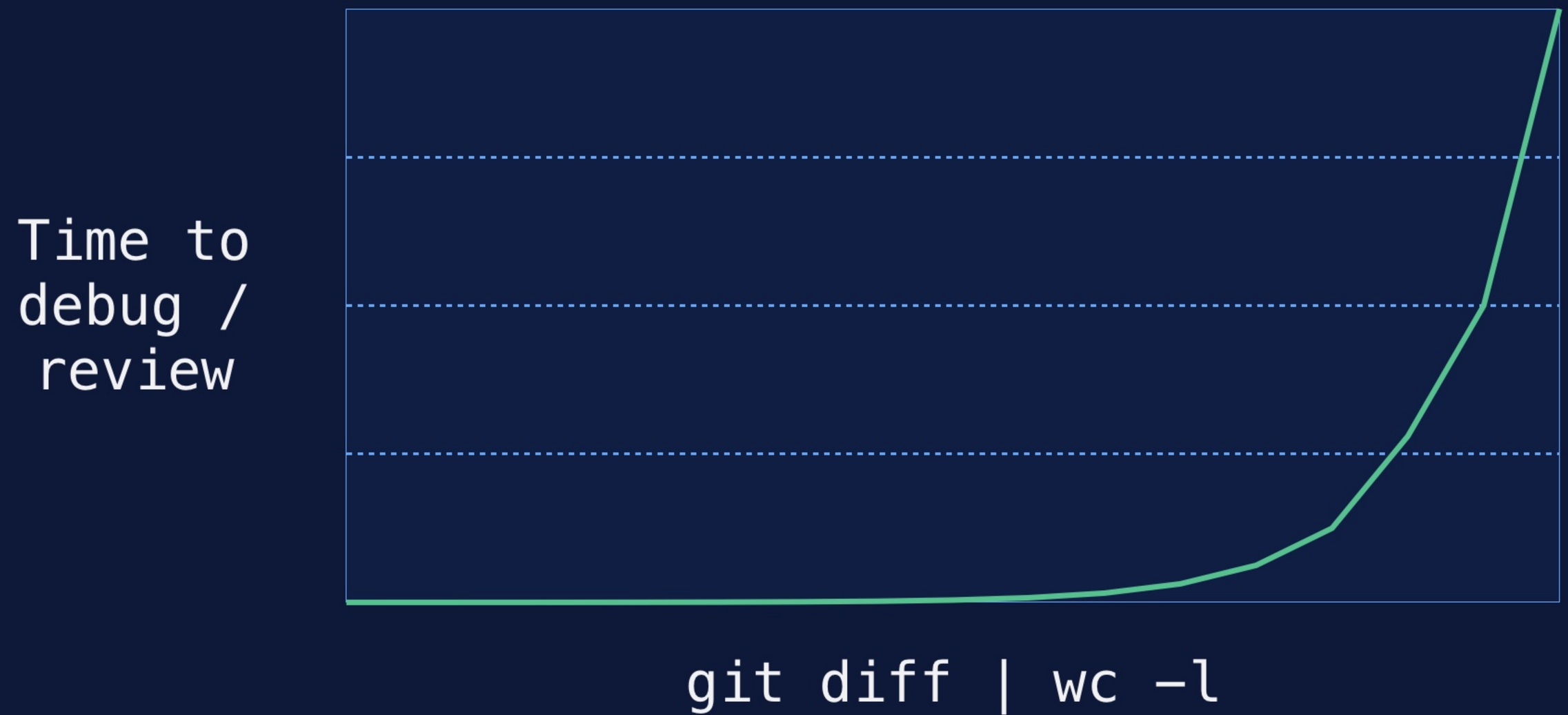






When you change a bunch of lines, in theory each of them might interact with every other thing you have.

The author of the commit can have a pretty good idea of which potential interactions are important, so coding is generally a tractable activity. It sort of works anyway.



So the amount of effort it takes to wrap your head around a changeset scales quadratically with the number of lines in it. Two small diffs tend to be easier to check for correctness than one large one.

If you’re trying to look at a diff that was deployed and figure out what went wrong with it, the problem is the same.

OBSERVATION

Our entire job is to ship many lines of code safely.

One solution to all of these problems would be to just avoid pushing lines of code. But we'd get fired pretty quickly if we did that. We have to ship a lot of code.

DEPLOYING OFTEN:

Minimizes the odds the deploy tooling has broken.

Deploying often means that we've exercised all of our tooling recently, and we can be confident that it works.

DEPLOYING OFTEN:

Forces us to have fast deploys.

Deploying sufficiently often means the deploy pipeline has to be fast. Which means there’s not a faster hacky way to deploy. Deploying often keeps us on the blessed path.

DEPLOYING OFTEN:

Minimizes the time until we can fix something broken.

That also means that when something breaks, we'll have a short path to fixing it.

DEPLOYING OFTEN:

Amortizes the odds that the code is broken.

Deploying often minimizes the chances that any given deploy is broken. We'll get a lot of little deploys through with no problems. If we deploy in huge chunks, we'll definitely have problems with them in production.

DEPLOYING OFTEN:

Means we grok the code
when we push it.

And when we encounter faults, we'll more clearly understand what we were trying to accomplish.

DEPLOYING OFTEN:

Reduces the time to debug if something breaks.

And we'll have an easier time figuring out what's broken if we just pushed a few dozen lines. The broken thing is something in that dozen lines. Not, as in other cases, some small thing in an epic pile of thousands of lines of code.

@mcfunley