Deployment Is Not Release

From Painful to Perfect

@garyfleming

TL;DR

TL;DR

— Deployment is painful,

TL;DR

— Deployment is painful,
— Feature Toggles are simple but powerful,

TL;DR

— Deployment is painful,
— Feature Toggles are simple but powerful,
— No fable,

TL;DR

— Deployment is painful,
— Feature Toggles are simple but powerful,
— No fable,
— Some cat pics.

Where Does it Hurt?

TODO toothache pic?

Often, when I start working with a new client, I ask people the same question. It's one that I find gets us quickly to some interesting areas to improve.

Now, I might formulate it slightly different, but the aim is to genuinely try to find things that people hate dealing with. The answers I get with alarming frequency all have something in common.

Where Does it Hurt, Developers?

Developers can take a while but often say it's doing a release. That thing they only do once a month, or three months, because it takes all weekend to ensure that it happens correctly. Who wants to spend all weekend at work?

Where Does it Hurt, Testers?

Testers often say its knowing what release and features are in each environment. They find it frustrating when they find apparent issues with systems to be dismissed with: "Well, that's not in that environment yet."

Where Does it Hurt, Product Owners?

Product Owners often say it's just getting things into the world. Because releases don't happen that often, it takes a long time to deliver the value they promised. Planning becomes difficult because there's a weird rhythm around releases: they take time, make people afraid to make changes that are too big.

Where Does it Hurt, Team?

The other thing that everyone said, in some form, was that it bothered them that they couldn't tell whether what they were doing would be useful. Sure, it'd pass the tests, and look about right, and go into production (eventually), but would it deliver the value they had hoped for? Is it what they really needed?

Gain Feedback. Deliver Value.

While I could do a full talk on the answers I've had to "Where Does It Hurt?", let's focus on these answers. People want to deliver things of value, but don't have the feedback they need to do that in a timely fashion. That's a problem.

Metathesiophobia

but underpinning this is Metathesiophobia

Metathesiophobia: Fear of Change

TODO evil clown pic?

... a fear of change. To deliver value and gain feedback, we need to release. Releasing causes pain. We've been hurt enough and don't want hurt again, so we delay it and our feedback along with it. So how do we break this cycle?

Deployment Is Not Release

We do it by making deploying software in a continuous and controlled way such that we can gain feedback without showing everyone, the norm.
That is, we make it so that Deployment Is Not Release.

Deployment Is Not Release

(Roll Credits)

Let's talk about how do to that. But first. Context.

Scenario: Context

— Mostly Co-located Team,
— Building Services (i.e. not products),
— Using Version Control,
— Some Trust.

This is a little more hands-on than many of my previous talks, but you don't
need much tech knowledge. For much of this talk I'm going to assume that:
* We're talking about a team who work near each other,
* Who trust each other to some degree,
* Who are building a service suitable for CD, as opposed to a project that might be more stochastic.
* you know what version control is,

Scenario: Problem

Imagine that you and your team have been asked to deliver several new features to your online store: the payment provider is being replaced; the user profile section is being integrated with some social media site; and the recommendation system is being tweaked. All these features are separate, but there are some shared points in the code.

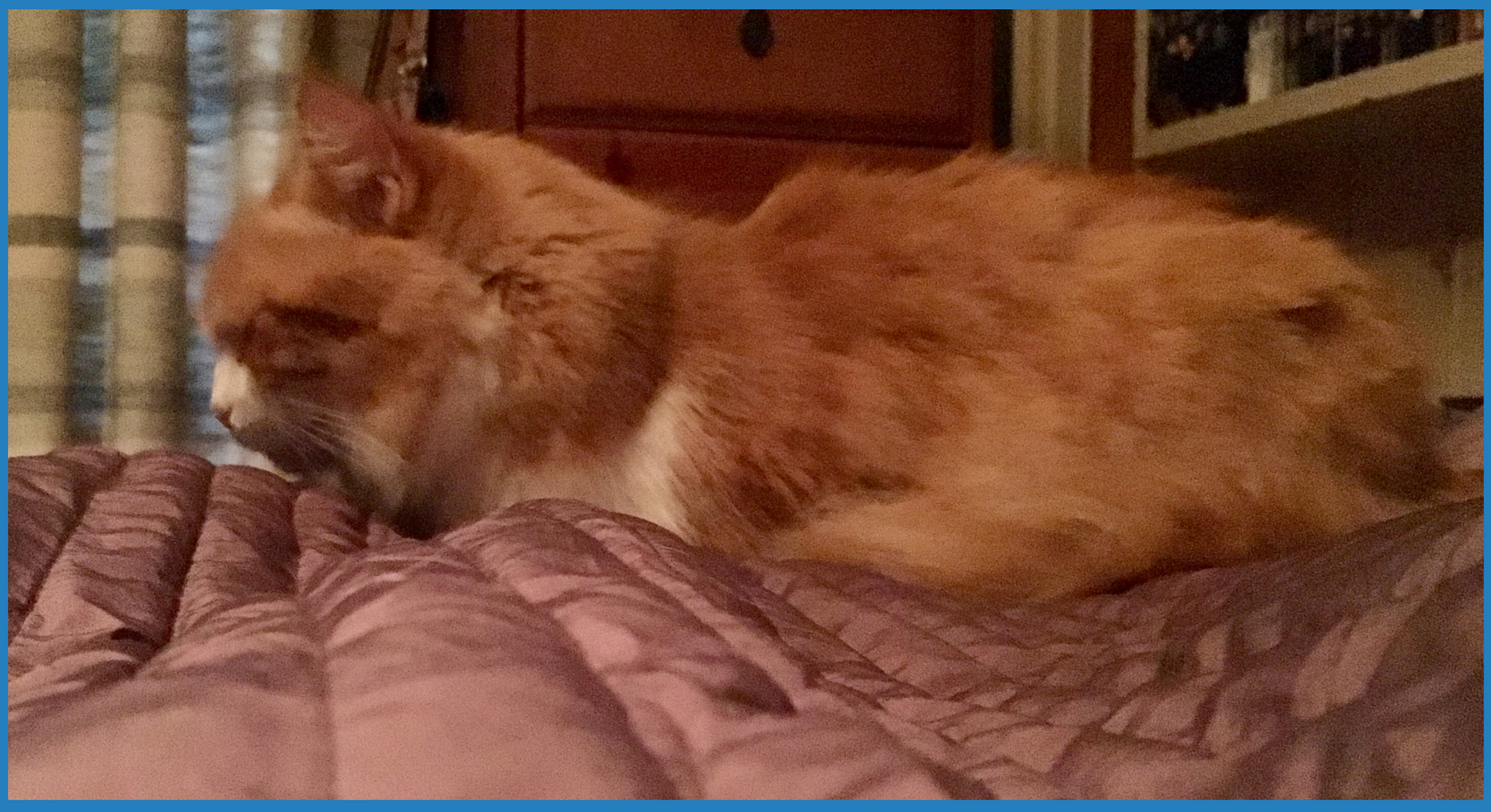
Release when?

The other part of the problem is that you don't know yet in which order these will be finished, so you don't know which one will go live first. That is, you need to be prepared for any of them to be live before the other ones. This leads to some interesting problems in arranging how you work together without conflicting. What's the solution?

Continuous Integration

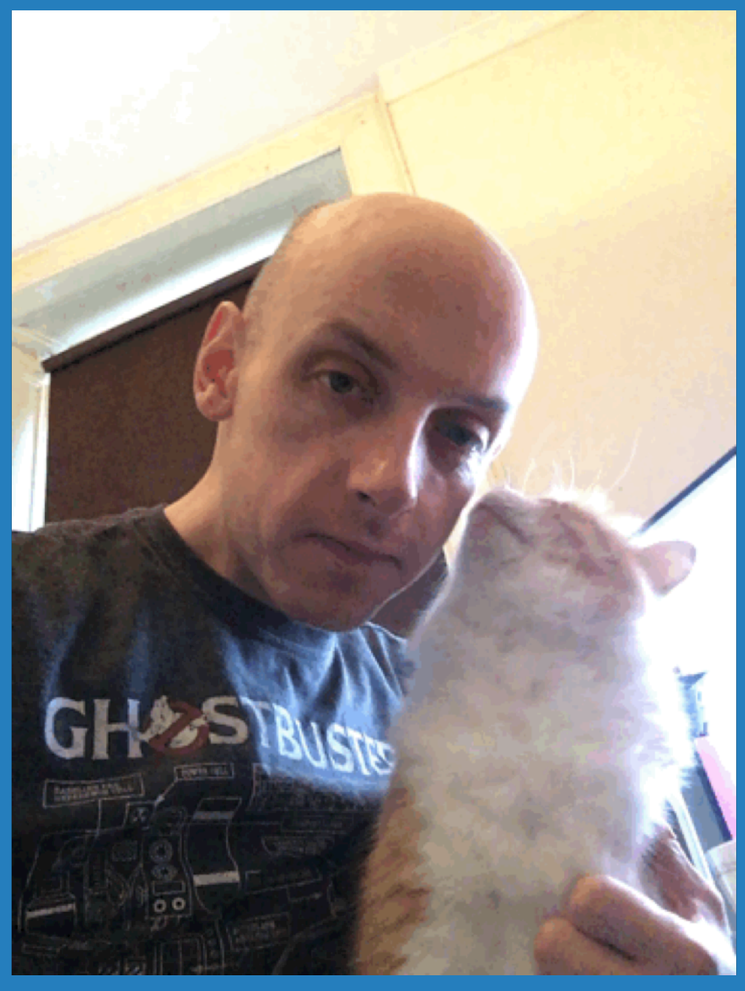
Before that, an important tangent: let's talk about Continuous Integration. I'm
increasingly of the opinion that the best way to really do Agile is by focussing on
Continuous Delivery of value. In a software context, that often means continuous delivery of features. That, in turn, heavily implies Continuous Integration.

The Cat Theory of CI Hardness



The Cat Theory of CI Hardness

The Cat Theory of CI Hardness



Continuous Integration - What?

Broadly speaking it's the notion that when we're working with other developers in a team it's a very good idea to bring our work together frequently so that we know that we're all still aiming for the same goal, without the potential for too many conflicts and issues along the way.

Continuous Integration - Four Steps

1. Update
2. Fix
3. Commit
4. Automate

To do, continuous integration well we do four things:
 * Before checking in to version control, we need to take everyone else's updates from trunk.
 * We need to fix any issues we see before we commit our changes. It is therefore *always* the responsibility of those who are making a change to ensure they work.
 * To ensure the continuous part, we commit frequently - at least once per day, preferably more often. I often commit safely multiple times an hour. When I can't do commit at least once an hour, it's often a sign that I'm working on something too big.
 * Finally, we run our build autonomously on a separate machine, so that we know everything works somewhere other than our own machine. This helps find issues.

Continuous Integration - Three Prerequisites

1. Test existing behaviour
2. Test new behaviour
3. Reliable, fast builds

To make those tasks possible we need to ensure three other things are always true:

 * Any behaviour we want our system to exhibit MUST have a test, so we can ensure that behaviour remains.
 * Any new behaviour we want our system to exhibit MUST also have a test, so we can ensure that behaviour remains in the future and that we get the behaviour we expect.
 * Our build as a whole must be fast and reliable. No-one will check the build is working before committing if that takes hours or may not work for reasons outwith their control.

Continuous Integration - Why?

- Communication,
- Less Conflict,
- Less Rework

Why do we do all this? Because continuous integration is a definitive and relatively unambiguous way for teams to communicate. If you use Continuous integration, you know immediately when your changes are causing a problem for some else and vice versa; as opposed to finding out weeks or months later. Find problems early is much better than finding them later and have to rework. You should absolutely have real-world conversations, early and often, to avoid potential conflicts, but there will always be ambiguity there. Committing tested code makes that much harder.

Continuous Integration - When?

If CI is about getting feedback quickly so we can work together better, how often should we be getting that feedback? Personally, I want to know about those conflicts quickly. At least every day, preferably far more often. More on that later.

Feature-Branching - The Idea

Now back to solving our problem of trying to deliver multiple features: one solution is called Feature-Branching, or sometimes just branching. The idea here is that for every feature that you want to deliver, you create a new branch in your version control system. All work for that feature is then done exclusively on that branch, and any changes on trunk are merged onto the branch (usually in branches, periodically). In the Github Pull-Request style, these merges are often further delayed by code reviews.

Feature Branching - Isolation

When a feature is complete, to release it, you just merge the branch back into your trunk, and the feature is there and is ready to be used. While developing it's been completely isolated so you know that it can't cause you any conflicts while it is on its own branch.

Feature Branching - Release

That's really nice because it lets you cherry-pick the features you want to release simply by controlling what you allow into your trunk. If you don't want to release it yet, well, leave it on the branch a bit longer. It also means that you can have as many parallel features in progress as branches, which surely is a good thing?

Problem: Feature Branching is Anti-CI

Well, no. Because if you're doing feature branching, you're not doing continuous integration and getting the benefits of doing that. Every branch is now spending weeks or months separate from the trunk, i.e not integrated, and the potential for conflicts increases rapidly with every day and every branch. You're no longer communicating.

"Continuous" does not mean "once, at the end", and "integration" does not mean "keeping everything apart until we can't"

Problem: Feature Branching - Manual Merging

Merging is also error-prone and manual. Because you're relying on people to bring together disparate threads of your system months after they're finished, they have to reason about multiple things that they may not have seen and may not fully understand.

Feature Branching - Summary

"Feature Branching is a poor man's modular architecture. Instead of building systems with the ability to easily swap in and out features at runtime/ deploy time, they couple themselves to the source control; providing this mechanism through manual merging."
— Dan Bardot

Bardot is hinting at our alternative: Feature Toggling.

Feature Toggling - A New Hope?

I've spent most of my time not talking about the subject so I guess I should fix that: by telling you what we don't do when feature toggling. We don't branch. All development happens on trunk, and never on a branch. That means we've immediately regained our continuous integration.

Feature Toggling - Isolation

The sharper minded amongst you are thinking, "Sure, you get continuous integration, but what about isolation? How do you develop features together safely?" That's the toggle part. For every piece of code that is either a work in progress or not ready to go live yet, you introduce an abstraction that hides away your implementation changes. You then build a mechanism, a toggle, that lets you switch between different implementations.

Feature Toggling - Additions

What that looks like: for an a new piece of functionality, it's very straightforward. Your toggle is going to be an on/off state. If the toggle is on, your new feature is included in the runtime. If it's off, it's not. So you can develop something on trunk that may or may not be ready for production yet.

Feature Toggling - Modification

For modifications, it's not that much harder: you take the existing version of the code and put an abstraction in front of it that provides a nice interface to both the new and old versions of your code. You then make all of your existing code use that interface. You write the new implementation, and have the toggle switch between the new and old at will.

Deployment Is Not Release

Doing feature toggling gives us an important capability. We can deploy our code many times a day. Rather than being the scary thing we do once every few weeks or months, it becomes routine/boring. That's a good thing.
The features we develop can be deployed but we can hide them from release until we're ready to show them to the world.

Strategies

Moreover, it gives us options for how we release features. I've described toggles
as being on/off, but that's not necessarily the case. You could use a toggling solution
that lets you switch them on for individual users, or roles, or IP addresses. You
could see how effective features are by switching them on for a small percentage
of users. 1% of users. Aged 18-35. In Edinburgh. Then 5%. Then all of Scotland etc
If it doesn't work out, switch it back off and most people never saw it.

Complexity

(NOT in the Cynefin sense)

Doesn't this mean we have lots of extra Complexity with things being on/off, or that our code is littered with if statements? Well, a little. Toggles for making updates should generally be short-lived: you use them to hide a feature, and after that feature is live and has been proven to work, you remove the toggle entirely. Don't let them build up.

Most of the Complexity Issues...

"Today's realisation: most of the issues people have with feature toggling are also there when branching, but easier to see when toggling."
— Me, pretty soon after giving this talk a while ago.

Also the alternative is ALSO complex, in a way that you can't easily see.

Back To Context

— Mostly Co-located Team,
— Building Services (i.e. not products),
— Using Version Control,
— Some Trust.

Having said all that, let's think about the context I laid out near the start again, and consider what happens when that's not right. The Github PR model makes more sense when you're building a product that makes sense to bundle versions (i.e. not CD), when you're building with other developers around the world that you don't necessarily trust to get it right, or when timezones and remote working mean that alignment is tricky.

What do we get?

So, hopefully, I've covered what it is. Let's talk a little bit about what different parts of the team get from using Feature Toggles.

What do we get as Developers?

It gives developers a good reason (if ever they needed one) to focus on technical excellence. If the code can go live at any time, devs need to be able to handle that smoothly and safely. That becomes a great reason to always have a well-built, well-tested codebase.

What do we get as Developers?

Ask Your Developers:

"If I were to push every one of your commits live today, what would you want to be in place to feel confident and safe?"

This might initially cause fear or panic, as discussed earlier. But it's a great opportunity to have a conversation about what developers need. You also want to have conversations with your product owners so you know when they want stuff off at first.

What do we get as Testers?

Make sure the devs give you a way of managing the toggles so you can test properly.
Have conversations so you know where different toggles might collide. Assume there are more than the devs tell you.

What do we get as UXers?

Play with strategies. Toggles let you do rolling A/B tests.
Use that capability to improve.

What do we get as Product Owners?

You don't need to manage deployments any more (if you ever did)
You can now truly own the product by deciding when things are ready, based on customer feedback. This is powerful.

What do we get as a team?

— Boring releases (GOOD!)
— Faster feedback, safely.
— Metathesiophobia removed.

As a whole, we get exactly what we set out to get at the start: the ability to do deployment frequently, in a boringly safe way, without actually releasing features until we're ready.
We get the power to experiment easily and safely so we can gain feedback quickly.
We lose our fear of change, because change is always happening and it is mundane.

Summary

Think about what this achieves: all of your team can work together and see exactly what's coming down the line in a way that is safe. If conflicts arise you know about them immediately and can resolve them before rework becomes painful, and you can keep changes isolated as long as you need. Compare that to the alternatives, and it's a clear advantage. Thank you.