corebytes
a fintech blog from coremont

coremont

Migrating to Continuous Deployment to Production

By Nick Owen, Head of Software Engineering

January 2024

Migrating to Continuous Deployment to Production

Continuous Deployment to production has been the goal of Coremont's technology teams over the past three years. Their aim has been to engineer away the time and energy spent on releasing software. Time and energy spent running manual test scripts, comparing diffs of data, seeking sign-offs and approvals and finding change windows. Abstracting away the idea of a release has freed our teams up to focus on innovation, quality and overall bigger, more interesting, challenges.

I'm often asked how frequently we release our software. I find this a tough question to answer for a number of reasons.

Firstly, there is some mysticism around this value. It's an industry meme (perhaps a tired one) that large banks suffer the worst. Recently I heard from someone who'd managed to automate through much of the red-tape by scripting the change control process to send email approval requests to stakeholders with appropriate test evidence attached. Another, more contemporary, pattern is to attend a meetup and hear about a smaller, more nimble, organisation that could lay down the correct principles on day one, and have since managed to scale out to thousands of releases a quarter.

Secondly, fundamentally – *I honestly have no idea what the answer is*. At any given time of day, the answer is not at my fingertips. It's very likely that the answer is greater than zero. But the important thing is that it doesn't matter. It's not about releasing all the time; it's simply that when you make a change, it will go to production. The question really becomes, "has anyone made a change today?". A reward in achieving a trusted continuous deployment pipeline is that we don't have to think about it again.

And thirdly, like all matters, context is key. There's no value to the number itself. The value is in how we got there. We're not a massive institution, but nor did we have the benefit of starting out with the end in mind. Here's how we did it.

Definitions

This article from Atlassian does a brilliant job explaining the differences between *Continuous Integration (CI)*, *Continuous Delivery*, and *Continuous Deployment (CD)* – I won't try to improve on it.

At Coremont we practise all of these, but this article is focused on how to evolve from *Continuous Delivery* to *Continuous Deployment*.

Premise

The starting point of this article is a system that would benefit from the investment of continuous deployment to production. At a minimum I would suggest:

- **The team already practises continuous integration**
  We use trunk-based development with GitLab as our git strategy. You don't have to do the same (e.g. GitHub flow can be close enough), but the strategy needs to align well with rolling commits forwards and back through the main pipeline.

- **Service deployment is automated and in source control**
  Terraform, Helm or good old bash. Whatever it is, it needs to be repeatable, stable and automated. It needs to be owned by the team, and changes to the deployment process need to be subject to code review like any other application change. Deployments need to be

testable and atomic. Vitally, you will need to be able to rollback to previous versions with minimal fuss.

- **Some continuous delivery, or deployment to nonprod environments, is present**
  If the system isn't continuously deploying to a nonprod environment (whether an ephemeral temporary environment or a persistent test environment) then you're probably aiming too high right now. Start small – having your changes roll into a test environment will still bring learnings and value.

- **There is already a good coverage and quality of automated testing**
  Continuous deployment is about automated deployment. The more manual checks you do or require, the more ground you have to cover to get there. At the least, the basics of the testing pyramid should be covered.

- **Separate the concept of feature release from code deployment**
  Octopus said it better than I will. The point is that in an emergency you don't want to be waiting for your deployment code to turn off a user feature. Start embracing feature flags. Deploy your new dormant features into production and toggle them on and off at runtime. Decouple your product team's release cadence from your development team's merge frequency.

If you don't have all of the above then you can still continue, but note you'll be embarking on a larger project. It took us years to get to this point alone.

How?

Culture

The conditions need to be right. The culture, both within the team and among the team's stakeholders, needs to accept the premise of continuous delivery and deployment. If your company, culture or team aren't even here then you need to start on winning over hearts and minds.

There are resources everywhere for this, but restricting myself just to one I'd recommend The DevOps Handbook a.k.a. the yellow bible.

Once you've convinced yourself, convince your team. Once your team's on-board, convince your manager. Work outwards, building consensus until you start to feel a critical mass. Go to meetups and hear why other teams have done this. For me it was important to be upfront at the start of the journey: this will not stop bugs getting into production. Bugs will still happen. The positives are that bugs should become easily revertible, or quickly fixable.

To this day I've not seen nor heard of any team who has regretted embarking on this journey, let alone gone back on it.

Where to start? Choose wisely.

The dev team will need to spend time analysing the weak points, shoring up the testing, building or changing pipelines and thinking deeply about their potential error states.

The system needs to be worth the investment. A very stable service that's released once a month may seem a safe place to start, but likely won't return the investment in a reasonable time frame. Similarly, a service whose blast radius would be felt across the product or business, with a lot of weekly change, may make stakeholders uncomfortable.

Analysis

Time to find the gaps. Here are some useful considerations:

- What is missing to make this a success?
- What have been the causes of any production rollbacks over the past year? Do any themes emerge?
- Which bugs have slipped through the existing processes, and what tests can be written to cover them?
- Where on the testing pyramid are you most lacking coverage? Is your dependency injection covered? Are there enough unit tests on the hottest code paths?
- Do you have all the right runtime signals? Do you have proper health checks and real-time alerting on key metrics. When the service goes wrong you need to know ASAP.
- How do your performance characteristics change as the number of deployments skyrocket? Will you be missing caches more often than not? Do you need to think about cache-warming, or distributed caching? If you run blue/green deployments and have two versions of your service running simultaneously will all your downstream connections handle twice the load?
- Can you deploy forwards and backwards without breaking your contracts? These can be contracts with other teams through an API, or within your own service with its datastore.

Note: not every bug comes with its own pre-built testing framework. Each service is different, and may require a bespoke approach. For example, maybe a bug only appears during a deployment of a distributed service, where the mixed estate of services interact with a relational database.

Now you've identified the gaps you're ready to treat this like any other project. Size the change, prioritise the work, and start finding time to execute.

Stay the course

The team is invested, stakeholders are bought in and work is underway. You may still be months, weeks or just days away from your target date. During this time more effort may be invested into this service than ever before.

At some point, there may be a setback. Maybe a new failure mode becomes apparent, and a new test framework needs to be bought, conceived or implemented. Other client work or deadlines may take urgent priority. A production incident may erode confidence in the work already achieved and the work yet to come.

It may be easy to say, but don't lose faith, and especially – do not rush. It's not a race to the finish line. The ethos is to calmly and steadily continue to knock down your impediments until the work is done.

Each new production incident, especially if release related, should be thoroughly examined against the existing body of work. If you're lucky, you'll see that one of the planned work items for this project would have covered this error. Otherwise, a new gap has been identified in your plan and you need to find a way to cover it.

Flick the switch

Getting this far is a victory in and of itself. By this point your service will have received so much attention that, if done well, it will hopefully be the best it's ever been.

Testing will have increased, so overall confidence will be increased. Production errors should be decreasing.

Turning on continuous deployment can be no different from all your future releases. The pipeline and code should already be in place so that the change in behaviour is a flick of a switch (deployments are not releases). Your pipeline should be able to continue to production via configuration. Make a plan for how to roll this out, and how to roll it back if it doesn't go well.

Choose a time that isn't critical to your business. Have a few small, trusted, changes ready to go so you and your team can see this in practice a couple of times before leaving the office.

Trust in the process, so that you don't have to think about it again.

Appendix: Our Stats

2020

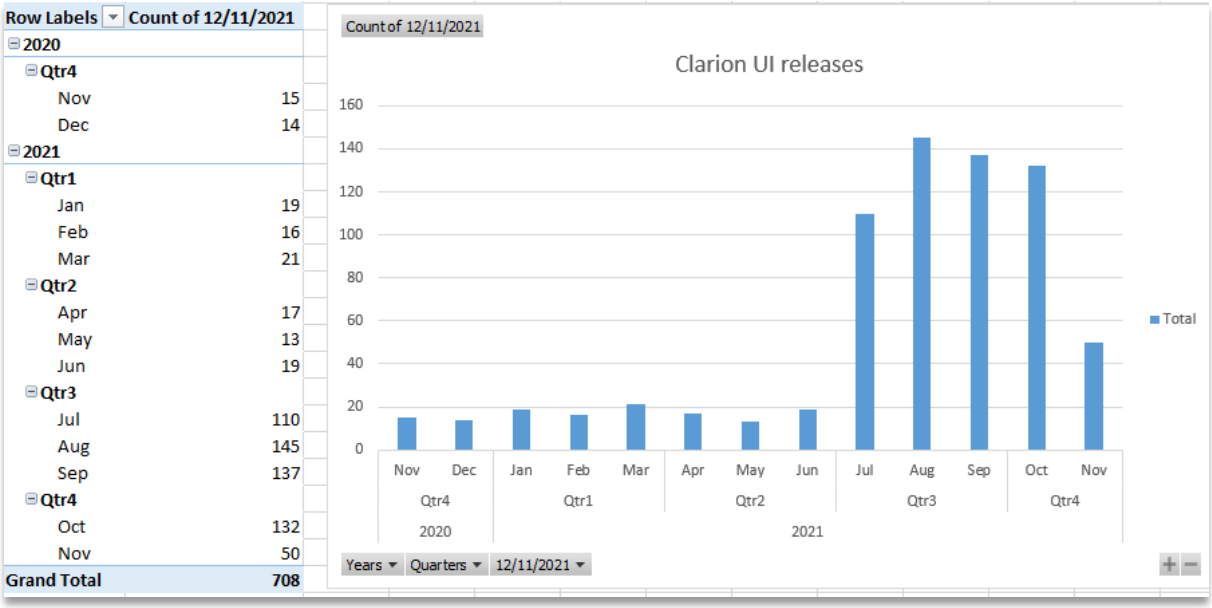
At the start of 2020, for the scope of this article at least, no service involved in our flagship product, *Clarion*, was continually released to production. Instead, we grouped our services together and, sometimes daily, attempted to release the entire crop in one go.

With some significant manual and testing effort, we managed to release *at least one service* on 300 days in the year.

In November of 2020 our first service, the API for Clarion's PMS UI, started continuously deploying to production.

2021

One year on, in November 2021, two more services had joined this API as continuously deploying: our UI and a permissions service.



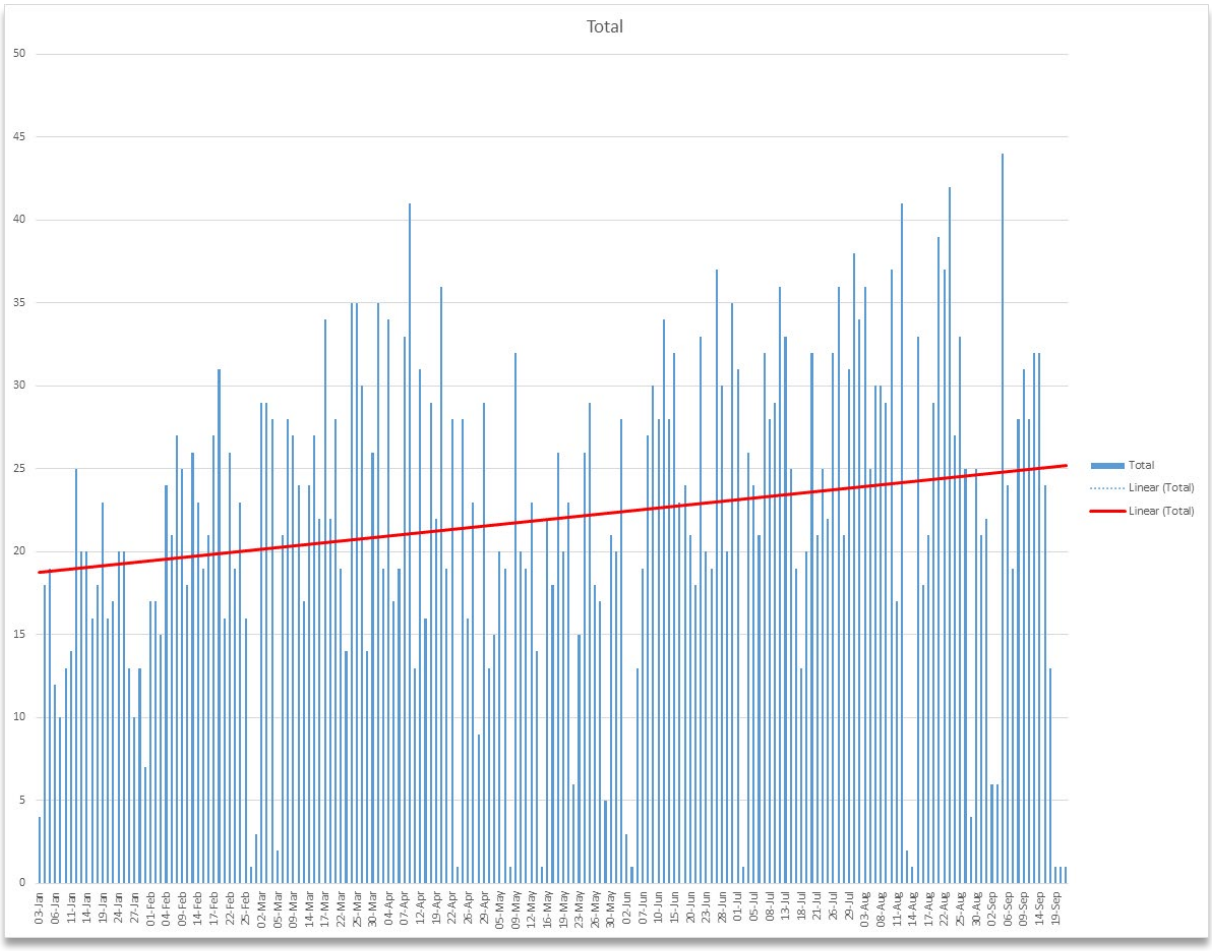
Row Labels	Count of 12/11/2021
2020	
Qtr4	
Nov	15
Dec	14
2021	
Qtr1	
Jan	19
Feb	16
Mar	21
Qtr2	
Apr	17
May	13
Jun	19
Qtr3	
Jul	110
Aug	145
Sep	137
Qtr4	
Oct	132
Nov	50
Grand Total	708

The metric we observed changed – instead it was now the sum of each service's releases, and we had crossed 2000.

2022

Three more services joined the gang, including our largest service, the distributed pricing grid. Our totals grew to over 5000.

The daily count became interesting – our biggest day was 44 releases from these six services.


Total
50
45
40
35
30
25
20
15
10
5
0
03-Jan
06-Jan
11-Jan
14-Jan
19-Jan
24-Jan
27-Jan
01-Feb
04-Feb
09-Feb
14-Feb
17-Feb
22-Feb
25-Feb
02-Mar
05-Mar
09-Mar
14-Mar
17-Mar
22-Mar
25-Mar
30-Mar
04-Apr
07-Apr
12-Apr
19-Apr
22-Apr
26-Apr
29-Apr
05-May
09-May
12-May
16-May
19-May
23-May
26-May
30-May
02-Jun
07-Jun
10-Jun
15-Jun
20-Jun
23-Jun
28-Jun
01-Jul
05-Jul
08-Jul
13-Jul
18-Jul
21-Jul
26-Jul
29-Jul
03-Aug
08-Aug
11-Aug
14-Aug
17-Aug
22-Aug
25-Aug
30-Aug
02-Sep
06-Sep
09-Sep
14-Sep
19-Sep
Total
Linear (Total)
Linear (Total)