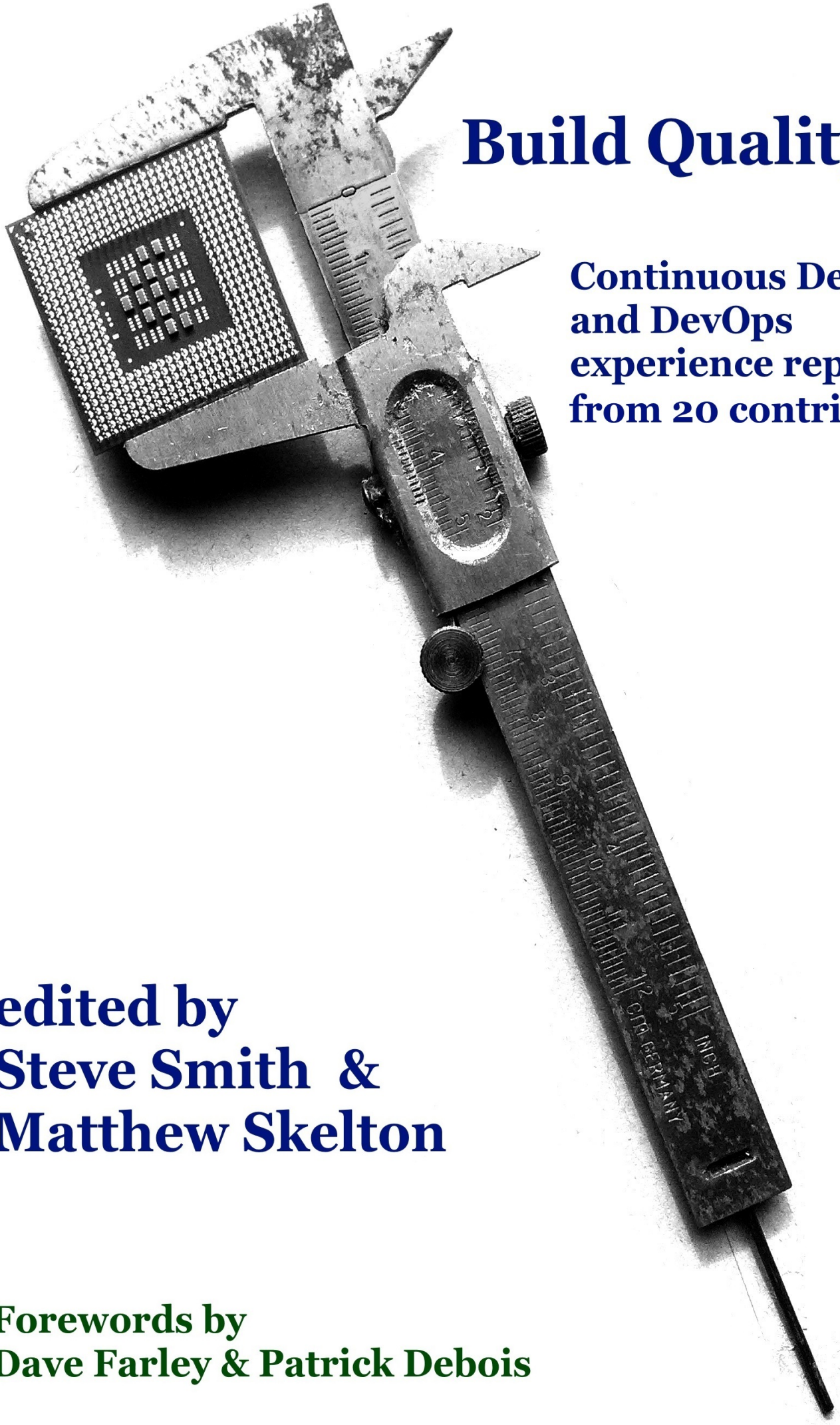
Build Quality In

Continuous Delivery and DevOps experience reports from 20 contributors



edited by
Steve Smith &
Matthew Skelton

Forewords by
Dave Farley & Patrick Debois

Build Quality In

Continuous Delivery and DevOps Experience Reports

Steve Smith and Matthew Skelton

This book is for sale at http://leanpub.com/buildqualityin

This version was published on 2018-07-22

ISBN 978-1-912058-57-0

Tweet This Book!

Please help Steve Smith and Matthew Skelton by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought the @BuildQualityIn book! www.buildqualityin.com #continuousdelivery #devops

The suggested hashtag for this book is #buildqualityin.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#buildqualityin

Also By These Authors

Books by Steve Smith

Measuring Continuous Delivery

Books by Matthew Skelton

Team Guide to Software Operability

Steve dedication: to my wife and daughter.

Matthew dedication: in memory of Mike G, who taught me about computers, coffee, and music.

Contents

Introduction . . . . . **i**

Contributors . . . . . **ii**

Donation to Code Club . . . . . **iii**

Continuous Delivery Foreword - Dave Farley **1**
About Dave 2

DevOps Foreword - Patrick Debois **3**
About Patrick Debois 3

Learning to dance to a faster rhythm - Chris O'Dell **5**
The opening sonata 5
The slow adagio 7
The dance of the minuet 10
The closing sonata 11
About the contributor 12

DevOps-ifying a traditional enterprise - Niek Bartholomeus **13**
Introduction 13
Organisation structure 14
Problems 18
Tactical solution: enhancing the existing communication flows 19
Structural solution: decentralisation 20
Summary 24
About the contributor 25

A testing transition from yearly to weekly releases - Rob Lambert and Lyndsay Prewer . . . **26**
The problem 26
Our vision 26
Who does the testing? 27
What approaches do we use? 27
The deployment process 28
Feedback loops 28
Challenges 29
The future 29
About the contributors 31

Delivering Continuous Delivery, continuously - Phil Wills and Simon Hildrew **32**
Why deploy so frequently? 32
How did we make it a reality? 33

RiffRaff, a tool for simple deployment . . . 33
Monitoring and alerting . . . 34
Why not Continuous Deployment? . . . 35
Roles and responsibilities . . . 36
About the contributors . . . 36

Making the world a better place - Marc Cluet . . . 38
Context . . . 38
High level DevOps . . . 38
Is it all about 'the Cloud'? . . . 39
Introducing DevOps . . . 39
Teaching an old dog new tricks . . . 42
In Conclusion . . . 43
About the contributor . . . 43

Staying Xtremely Unruly through Growth - Alex Wilson and Benji Weber . . . 45
Introduction . . . 45
Testing in Production . . . 46
Co-ordinating Shared Infrastructure . . . 47
Making things visible . . . 47
Services/Versioning . . . 48
Delivering major design changes incrementally . . . 49
Conclusion . . . 50
About the contributors . . . 52

We need two DevOps resources, please - Anna Shipman . . . 53
Introduction . . . 53
How we built a DevOps culture at GDS . . . 53
GDS and the GOV.UK website . . . 54
Our DevOps culture . . . 55
The challenges we've faced . . . 61
What can someone reading this take back to their organisation? . . . 64
Coda . . . 65
About the contributor . . . 65

Process kick - Amy Phillips . . . 66
Needing a change . . . 66
Making the change . . . 68
Seeing Progress . . . 68
Dealing with uncertainty . . . 69
Keeping the Peace . . . 70
Did it work? . . . 70
About the contributor . . . 71

Scrum for Ops teams - James Betteley . . . 72
About the contributor . . . 76

DevOps in the mix - John Clapham . . . 78
Introduction . . . 78
An Unsustainable Pace . . . 78
A Pause For Thought . . . 79

First steps . . . 80
Momentum . . . 81
A Tipping Point . . . 81
Reflections . . . 81
About the contributor . . . 82

You write it, you support it - Jennifer Smith . . . 83
Understanding infrastructure . . . 83
Feedback loops . . . 84
Feedback loops from running systems . . . 84
Shortening the feedback loop: "You write it, you support it" . . . 85
Collaboration . . . 86
Trying this yourself . . . 87
Credits . . . 88
About the contributor . . . 88

Continuous Delivery across Time Zones and Cultures - Sriram Narayanan . . . 89
Introduction . . . 89
The Origin of the Build and Release Team . . . 89
Taking stock . . . 90
Performant environments and reduced functional test times . . . 90
Network issues, bandwidth issues, and network latencies . . . 90
Reducing build times . . . 90
Providing predictable environments . . . 91
Improving test predictability and stabilizing tests . . . 91
Monitoring builds . . . 91
Antipatterns . . . 91
The pipeline structure . . . 92
Lessons we learned for effective Continuous Delivery . . . 93
Remaining challenges . . . 94
Build and Release, or Infrastructure Engineering? . . . 94
About the contributor . . . 94

DevOps in an Enterprise environment - Jan-Joost Bouwman . . . 96
Introduction . . . 96
The process organisation . . . 97
Starting the journey . . . 100
Ambitions for the future and problems we are facing . . . 103
About the contributor . . . 106

Trust, configuration management, and other white lies - Martin Jackson . . . 108
How do you do Continuous Delivery and DevOps within the UK Government? . . . 108
How can trust be weakened in CD and DevOps in general? . . . 109
How have we established trust within the UK Government? . . . 111
Conclusion . . . 117
Song title references . . . 118
About the contributor . . . 119

Avoiding the pendulum swing - Rachel Laycock . . . 120
Context . . . 120

Conway's Law . . . 120
No more dependencies! . . . 121
5 years passed later . . . 121
Modularise all the things! . . . 121
Death by autonomy . . . 122
Stop the pendulum swing! . . . 123
Parting thoughts . . . 124
About the contributor . . . 124

Communicate to collaborate - Matthew Skelton . . . 125
Context . . . 125
Problems and causes: mental and financial models, execution, and communication . . . 126
Treat the build system as production . . . 127
Improving the software release process . . . 131
Fostering communication, collaboration, and trust . . . 143
Raising awareness within teams and across the organisation . . . 143
Results . . . 147
Lessons learnt . . . 148
About the contributor . . . 150

34 days - Steve Smith . . . 151
Introduction . . . 151
Year 1 . . . 152
Year 2 . . . 157
Year 3 . . . 162
Reflection . . . 167
About the contributor . . . 170

Contributor Q & A . . . 171
How did you get involved in Continuous Delivery / DevOps? . . . 171
What do you see as the biggest advantage of Continuous Delivery / DevOps? . . . 173
What do you see as the biggest challenge in Continuous Delivery / DevOps? . . . 175

Version History . . . 178

Publisher details . . . 180

Introduction

Build Quality In is an e-book collection of Continuous Delivery[1] andÂ DevOps[2]Â experience reports from the wild.

Since the inaugural DevOpsDays[3] event was hosted by Patrick Debois in 2009 and the "Continuous Delivery[4]" book by Dave Farley and Jez Humble was published in 2010 the interdependent disciplines of Continuous Delivery and DevOps have enjoyed rapid adoption and already attained mainstream status. Creating an automated deployment pipeline and increasing collaboration between Development and Operations teams offers tremendous benefits to an organisation, with the potential for improvements in quality, lead times, and operability that will reduce costs and increase product revenues.

However, Continuous Delivery and DevOps are *hard*. In many organisations the creation of an automated deployment pipeline is impeded by significant technology challenges, and encouraging Development and Operations teams to work together can seem impossible. How do you automate an unversioned release script run from different unversioned servers for years? How can siloed Development and Operations teams work together when one party is incentivised by features and the other by stability?

We have seen Continuous Delivery and DevOps work in the wild. Steve worked on the greenfield LMAX pipeline and the brownfield Sky Network Services pipeline, and has seen how rapidly releasing software can improve quality and reduce risk. Matthew has worked with over 25 organisations to help them understand and introduce DevOps practices and Continuous Delivery. Furthermore, at the monthly London Continuous Delivery meetup[5] and the annual PIPELINE conference[6] we hear how other practitioners have applied the principles and practices of Continuous Delivery and DevOps to their own organistions, and regardless of results there are always points of interest and lessons to be learned.

We want to help others on their Continuous Delivery and DevOps journey, by sharing the experiences of those who have been there before - what worked, what didn't, and the highs and lows of trying to build quality into an organisation. We also want to raise the profile of other practitioners in this area, and contribute in some small way to improving diversity within the IT industry. We hope **Build Quality In** helps you on your own journey.

We would like to thank all of our contributors plus Dave Farley and Patrick Debois for their all-important contributions to this book. Thanks also go out to Kevlin Henney for the initial advice, London CD for the encouragement, Peter Armstrong and the Leanpub team for the help, and Laura Kirsop and the Code Club team for the inspiration.

Finally, we would like to thank our long-suffering families. We couldn't do this without you.

Steve Smith and Matthew Skelton, 2014

[1] http://www.continuousdelivery.com
[2] https://en.wikipedia.org/wiki/DevOps
[3] http://www.devopsdays.org
[4] http://www.amazon.co.uk/dp/0321601912
[5] http://londoncd.org.uk
[6] http://pipelineconf.info

Contributors

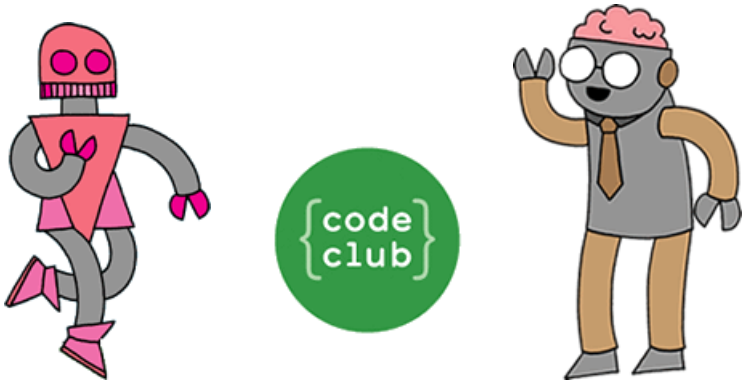
We have an incredible group of Continuous Delivery and DevOps practitioners, who have freely shared their own first-hand experiences in this area.

- Alex Wilson and Benji Weber
- Amy Phillips
- Anna Shipman
- Chris O' Dell
- James Betteley
- Jan-Joost Bouwman
- Jennifer Smith
- John Clapham
- Marc Cluet
- Martin Jackson
- Matthew Skelton
- Niek Bartholomeus
- Phil Wills and Simon Hildrew
- Rachel Laycock
- Rob Lambert and Lyndsay Prewer
- Sriram Narayanan
- Steve Smith

Thanks to all our contributors!

Donation to Code Club

We are donating 70% of author royalties to Code Club[7] - a not-for-profit organisation that runs a UK-wide network of free volunteer-led after-school coding clubs for children aged 9-11.



We passionately believe that diversity within the IT industry must improve, and efforts must start in our schools. From under-representation of women in STEM subjects[8] to the ever-present gender imbalance, from startup sexism scandals to discriminatory recruitment practices, the IT industry is in crisis right now, and we need to invest in the next generation of IT knowledge workers to ensure these problems do not continue.

Given Leanpub's royalty rate of 90% minus 50 cents[9], buying our book at the minimum price of $20.00 will result in a donation of $12.26. Buying at the suggested price of $25.00 will provide a donation of $15.40.

(90% * 20.00) - 0.5 = $17.52 total royalties (rounded up)

70% * 17.52 = $12.26 royalties to Code Club

(30% * 17.52) / 2 = $2.63 each to authors

Thank you for buying **Build Quality In**. As well as providing a range of in-depth Continuous Delivery and DevOps stories, your purchase will contribute to children learning to code and general skills such as problem solving and collaboration regardless of their gender or ethnicity.

[7] http://www.codeclub.org.uk
[8] https://en.wikipedia.org/wiki/STEM_fields
[9] http://blog.leanpub.com/2012/12/charity-royalties.html

Continuous Delivery Foreword - Dave Farley

Dave on Twitter: @davefarley77[10] Dave's blog: davefarley.net[11]

Continuous Delivery is a hot topic. Though comparatively new as a process, the genesis of this more rational approach to software development is diverse and spread through the experience of practitioners in the field.

What businesses really want of us as software developers is that we allow them to have an idea, get that idea into the hands of our users, figure out if the idea works, and react to the understanding that we gain from this "experiment". Continuous Delivery focusses on that feedback cycle and attempts to maximise it's efficiency. When we do this we write better software that tends to please our users.

It has taken experienced software practitioners decades to learn and refine these lessons. Over the years we have tried and failed with various approaches to solving this tough problem of realising business value in software efficiently and with high quality. We now think that we have an answer.

Continuous Delivery works because it is rooted in a more scientific approach to problem solving. We want to ensure that we operate our projects so that we can try out new ideas, establish feedback loops, reflect on the outcomes of our actions, and react to what we discover. This process is a highly disciplined, informal method of software development that focusses on trying to make the work that we do more verifiable. It applies an iterative, agile process of automation and sophisticated configuration management to steer our work. It is based on a more empirical approach. Continuous Delivery is a process that finally enables the organisations and businesses that fund our software development to be more experimental.

This approach works because it is, at heart, an application of the scientific method to the software development process. Since the scientific method is the most effective problem solving technique that humankind has ever invented then it isn't surprising to find that it works for the difficult problem of software development.

Organisations that adopt these techniques write higher quality software more efficiently. Such organisations are also more reactive to change and flexible in delivery and execution.

The only drawback is that this is not a simple process to adopt. It requires ingenuity, focus and courage. It usually requires changes to the culture of the organisations in which it operates and it challenges many traditionally held beliefs and working practices.

Adopting Continuous Delivery is not easy and it is not only a software development team effort. It will change the way that your business operates and interacts, for the better. This is not a trivial undertaking and it is a difficult path. However, the benefits are so pronounced that many companies have made this transition and none that we know of would willingly revert to the way that they worked before.

We believe that Continuous Delivery allows us to establish our approach to software development on a more empirical, more rational footing. This book captures the experiences of some seasoned practitioners and reports on their experiments and experiences.

This book is intended to help to speed-up your learning process. We hope that some of the experiences described here may help you to avoid some of the pitfalls and navigate to the high-ground. It doesn't mean

[10] https://www.twitter.com/davefarley77
[11] http://www.davefarley.net

that the transition of your development organisation and your business will be simple. It doesn't mean that you shouldn't continue to experiment and learn. Once you begin with Continuous Delivery you never stop learning and improving, but hopefully it will help you to at least avoid some of the mistakes that we have already made.

I hope that you enjoy this book, and I hope that the experiences of some of these experts in the field will help you make to make your transition to Continuous Delivery an easier one.

About Dave

Dave Farley *is co-author of the Jolt award winning book "Continuous Delivery[a]". He has been having fun with computers for over 30 years. Over that period he has worked on most types of software. He has a wide range of experience leading the development of complex software in teams, large and small. Dave was an early adopter of agile development techniques, employing iterative development, continuous integration and significant levels of automated testing on commercial projects from the early 1990s. More recently Dave has worked in the field of low latency computing developing high performance software for the finance industry. Dave currently works for KCG Ltd.*

Dave Farley

[a] http://www.amazon.co.uk/dp/0321601912

DevOps Foreword - Patrick Debois

Patrick on Twitter: @patrickdebois[12] - Patrick's blog: jedi.be[13]

Within the DevOps commununity there is a well-known acronym (first mentioned by John Willis) called CAMS[14]: 'Culture, Automation, Measurement and Sharing'. Many things have been written about the first 3 parts, and yes they are important. Sharing is often taken for granted, but it isn't:

- It takes **effort** to formulate the things you do, and put them in perspective.
- It takes **courage** to explain both the good, the bad, and the ugly in public.

Sharing has been a major part of the success of DevOps: people tweeting links, writing blogposts, organizing conferences, writing books - all under the #devops hashtag. This has allowed the community to learn from each other, to expand ideas, to finetune existing ideas.

In the old days, apprentices travelled from place to place to bring back new ideas to their guilds. They knew it was important to liberate themselves from their situation to get new perspectives, or even being able to see the problems in the right perspective. In the same vein, the book *Build Quality In* brings together stories from people who have been on a DevOps journey. The stories are not prescriptive so don't expect a 'DevOps steps 1, 2, 3' formula. They represent the learnings of the authors within their situation. Some things might be translated directly to your work situations, others will require you to rethink your strategy.

The most important takeaway is that you are not alone on this journey and that you should actively reach out to others to learn from, and that's what this book is all about.

I look forward to reading your blogposts, tweets, or even your own book. But first of all, start reading *Build Quality In* and get inspired!

About Patrick Debois

> ***Patrick Debois*** *is a developer, manager, sysadmin, and tester. He first presented concepts on Agile Infrastructure at Agile 2008 in Toronto, and in 2009 he organized the first 'DevOpsDays'. Since then he has been promoting the notion of 'DevOps' to exchange ideas between these groups and show how they can help each other to achieve better results in business.*

[12] https://twitter.com/patrickdebois
[13] http://www.jedi.be/blog/
[14] https://www.getchef.com/blog/2010/07/16/what-devops-means-to-me/

Patrick Debois

Learning to dance to a faster rhythm - Chris O'Dell

Chris on Twitter: @ChrisAnnODell[15] - Chris' blog: blog.chrisodell.uk[16]

7digital[17]

Timeline: August 2010 to July 2014

7digital's mission is to simplify access to the world's music. They do that by offering a proven, robust and scalable technology platform that brings business and development agility. Long-lasting relationships with major and independent record labels and a strong content ingestion system has brought their catalogue to over 25 million tracks and counting.

More than 250 partners use 7digital's music rights and technology to power services across mobile, desktop, cars and other connected devices. Their own music store (www.7digital.com) is localised for 20 countries, with apps available for all major operating systems.

Founded in 2004 in London's Silicon Roundabout start-up scene, 7digital now employs more than 100 people, of which roughly half are members of the Technology Teams, and they have offices in Luxembourg, San Francisco, New York, and Auckland. 7digital serves on average 12,000 requests per minute through the API with an average response time of 120ms. 7digital handle 3 million music downloads per month on average and can handle 22 millions streams per month serving petabytes of data.

The opening sonata

7digital aims to simplify access to the world's music. This is done via a robust, scalable, music platform powered by a flexible API. Of course, it wasn't always this way.

In 2004 7digital was born as a two man startup in the Shoreditch area of London, before it was trendy. It was a web-based reseller of digital music - MP3s, ringtones and even some video clips. This was a time after Napster had peaked and iTunes had started to dominate the market. Starting a company selling digital music was considered madness. Regardless, the little company sold DRM-free music direct to consumers and via white labelled miniature web stores.

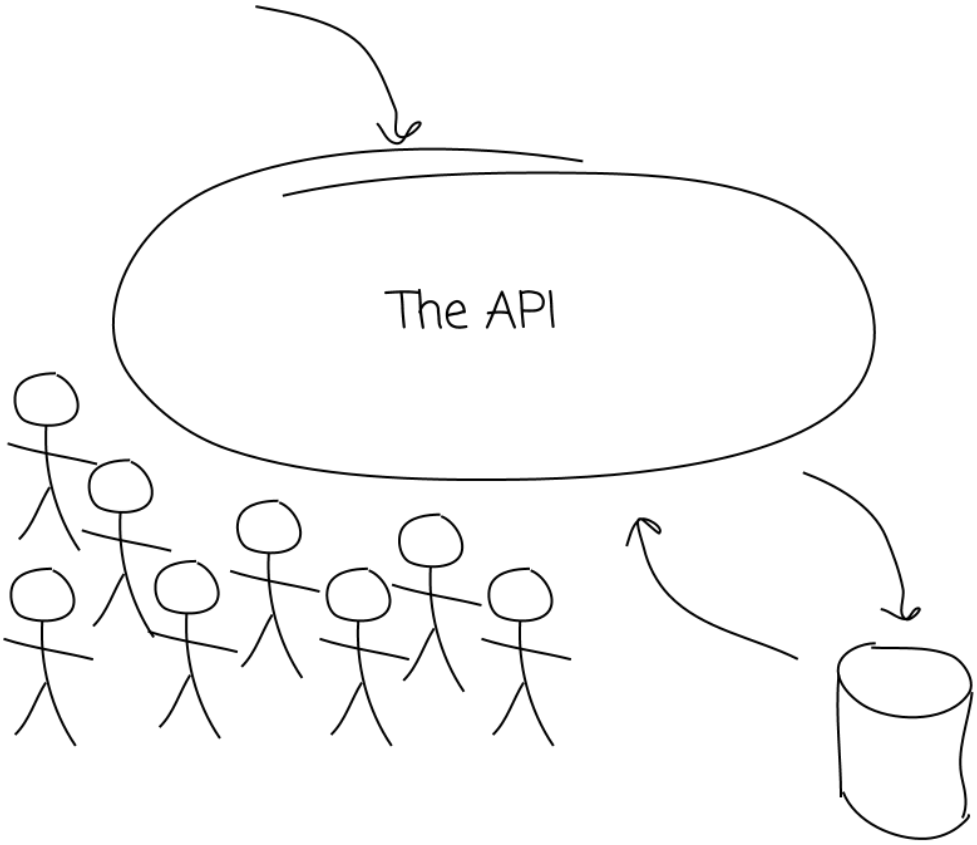
7digital also sold the collated music metadata of their catalogue to clients allowing them to build their own stores with the music files being supplied by 7digital. This data was provided in the format of large, ever increasing, CSV files. One client did not wish to receive the full CSV files, and asked for a way they could query and retrieve music metadata whenever they needed it. What they wanted was a web based API.

A single Asp.Net WebForms application was created. It was built using shared libraries which already existed to serve the consumer-facing website and the white labelled stores. This decision made a lot of sense at the time as the application's purpose was to simply expose existing functionality via the web. This also meant that the API shared the same database as all of the other applications.

[15] https://twitter.com/ChrisAnnODell
[16] http://blog.chrisodell.uk
[17] http://www.7digital.com

The application was developed with testing in mind, not exactly test driven, but there were end to end tests. These covered the small amount of functionality which the application provided and any new functionality was added with more end to end tests.

This approach served that client's purpose very effectively, and soon enough other clients gained access to it. The API grew gradually as each new client brought their own needs. The size of the test suite increased and the team supporting it also grew.



A rough diagram of the API's architecture

[18]

All applications were set up to run Continuous Integration using a shared TeamCity server. Each commit would trigger a build and a run of the full test suite before deploying to a pre-production server on success. With most of the test suite being end to end tests the time taken to run the suite increased along with the size of the codebase. Before long it was taking over an hour to get feedback, by which time the developer had lost context and possibly moved onto some other task.

Features were added, bugs crept in, and load increased whilst performance decreased. Time to add features sky rocketed, and the development team were treading on each other's toes to make changes.

With the API fast becoming the central part of 7digital's platform, we realised we had to take a step back, review our current approach and make an architectural change. We realised that we needed to split the monolithic application into smaller, more manageable products and by extension smaller, more focussed teams.

[18] https://speakerdeck.com/chrisann/evolving-from-a-monolithic-to-a-distributed-public-api

As a small company in a fast moving industry we couldn't afford to stand still. We could not take the time to develop a new version of the platform in parallel as a separate project. The changes had to be made to the existing application - we had to evolve it.

First we needed to get the current situation under control.

The slow adagio

When running a test suite takes over an hour, developers will start to employ a range of tactics for shortening the feedback loop. One example is to only run the obviously related tests on their local machine after making a change, thus leaving the full suite to be run by the Continuous Integration server upon commit.

This tactic relies on the developer knowing which tests are relevant and also remaining focussed whilst the full suite runs - it's tempting to assume the work is complete when you've run all the 'relevant' tests.

The end to end tests also suffered from fragility and 'bleed' by requiring the data in the database to be in a particular state. We would experience flaky tests that seemed to fail for no reason other than the order of execution.

Another tactic employed was the existence of a 'golden database backup' which contained the expected data for the tests to run. It was a large backup which could not be reduced in size due to the tangled and unquantified actions of the end to end tests. It would be copied to a new starter's machine like a rite of passage on their first day.

The above practices sound ridiculous, and they are, but you must realise that these things happen gradually - a single change or test at a time. As with the 'golden database backup' the pain is most evident when a new developer joins the team and the time it takes for them to get up and running is far longer than desired.

We knew this was a problem and that we needed to tackle it, but with such a large scope it was difficult to pin down. We took the approach that when working in a certain area you would review the associated tests, retain the main user journeys as end to end tests, and push the edge cases down to integration and unit tests. The edge cases included scenarios such as validation and error handling, which could be more easily tested closer to the implementation.

As the application was built with ASP.Net WebForms testing other than end to end tests was extremely difficult as the presentation and logic layers were deeply intertwined. Also the HTTP Context cannot easily be abstracted away, something which Microsoft made easier in later frameworks such as ASP.Net MVC. We decided to refactor every WebForm into a Model-View-Presenter pattern such that the WebForm itself did as little as possible and the business logic was pushed down into a Presenter class. The Presenter took only the elements it required from the HTTP Context and returned a Model which the WebForm bound to. This allowed us to unit test the business logic in the Presenter without needing to invoke the full ASP.Net lifecycle.

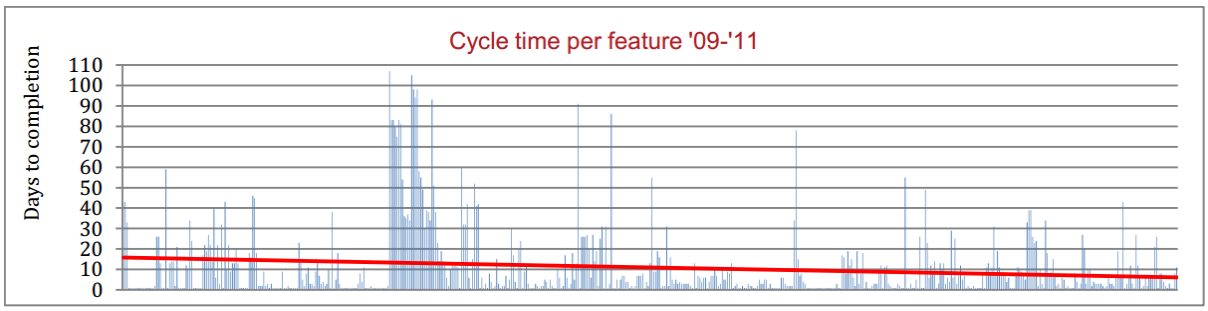
These changes significantly extended the time it took to fix a bug or add a feature and there were times when a refactoring was considered large enough to be tackled as its own work item. These items would be labelled as Technical Debt and prioritised in the backlog alongside the rest of the items. We had the full support of our Product Manager who had seen and understood the impact which the poorly performing tests had on our productivity, the platform's stability and our ability to deliver.

In our team area we had a small whiteboard where we noted down sections of code that we felt needed attention and we would regularly hold impromptu discussions around this board. This kept us focussed

on the goal even when the changes seemed impossible and morale was low. Crossing items off the board was a reminder of our progress and a source of pride.

Work items were tracked on a simple spreadsheet where we entered the date we started development and the date it was done. Our definition of Done was when the code from the work item had been released to Production. Rob Bowley[19], VP of Technology at 7digital, performed some analysis on this data which he published in a report in May 2012[20] and a subsequent report in 2013[21].

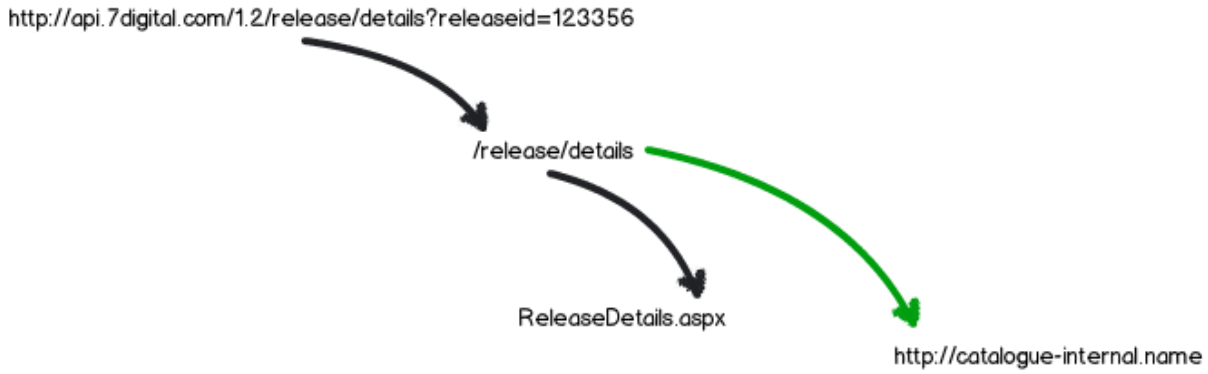
The interesting findings from the report show the team's cycle time during the period of refactoring greatly increased. The chart below shows a large spike where work items were taking more than 80 days to complete.



Cycle Time per feature 2009 to 2011

[22]

To enable the move to a Service Oriented Architecture a feature was added to the API codebase whereby incoming requests could be redirected to another service - an Internal API. The routes were configurable and stored in a database. The API would pattern match against the request URL and decide whether to handle the request itself or to pass it along to an Internal API.



API Routing based on pattern matching the request url

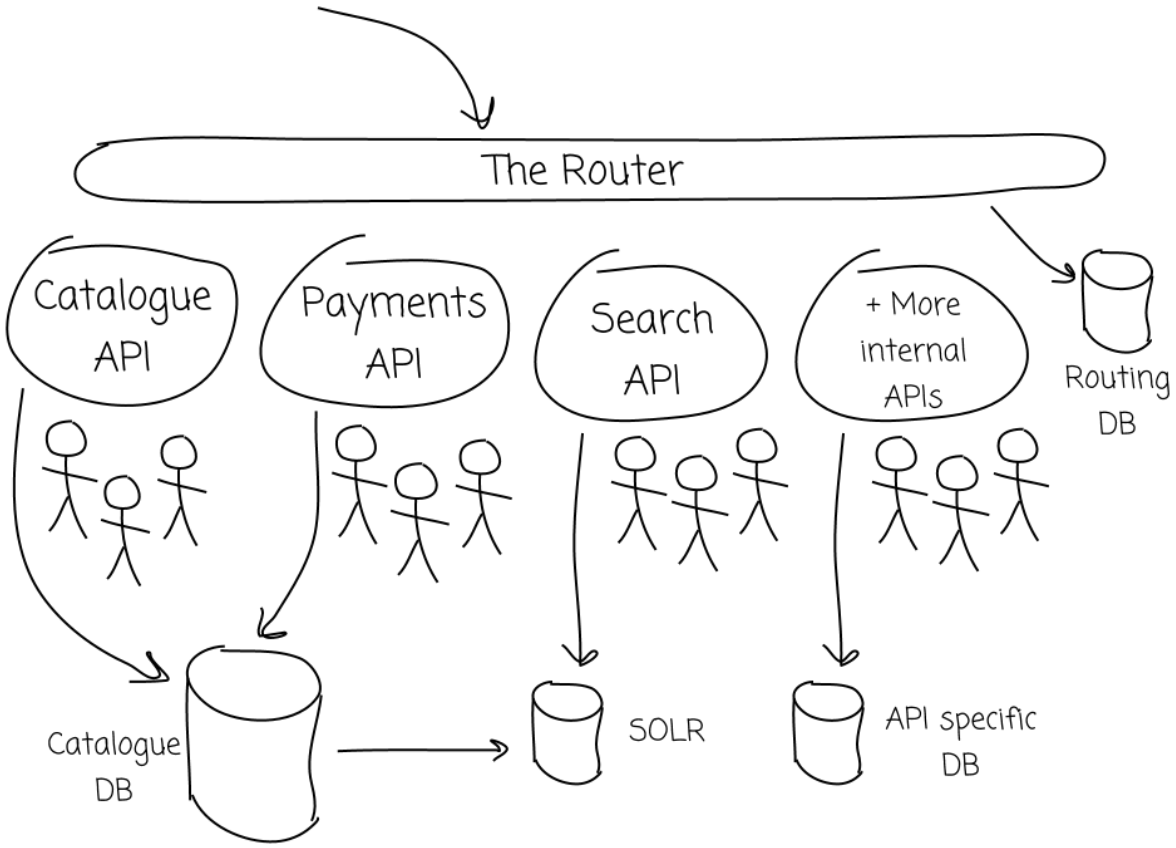
[23]

With the API acting as a routing façade we were able to carve out chunks of the functionality along domain boundaries. Internal APIs were created for Payment Processing, Catalogue Searching, User Lockers (user access to previous music purchases), music downloading, music streaming and many other domains.

[19] https://twitter.com/robbowley
[20] http://developer.7digital.com/blog/development-team-productivity-7digital
[21] http://developer.7digital.com/blog/7digital-development-team-productivity-report-2013
[22] http://developer.7digital.com/blog/development-team-productivity-7digital
[23] https://speakerdeck.com/chrisann/evolving-from-a-monolithic-to-a-distributed-public-api

In all cases the changes were extremely gradual and took years of work, with a single route being replaced at a time. Some were rewritten completely in new frameworks whilst others were first carved out by duplicating the existing code as a new project and rewriting it separately from the API. Each domain called for a different approach. For example, the Search functionality was rewritten to use SOLR as a more appropriate datastore, while the Purchasing functionality was cut out as-is to isolate the functionality and make it easier to understand and test before attempting to rewrite it.

The development teams also split apart from the API team into domain focussed teams: a Payments Team, a Search Team, a Media Delivery Team and so on. Each team was now able to focus on a smaller subset of the overall platform, and to operate as mostly independent projects. With the original API now a façade each team could release almost all changes independently and without need for co-ordination between teams.



A rough diagram of the SOA architecture of the API Platform

[24]

This separation allowed the teams to devise their own build and deployment scripts and finally move away from the now bulky Rake scripts. The Rake scripts were originally created to be a standard way of managing build, testing and deployment. Over time, features and exceptions had been added to them, eventually making them unwieldy, fragile and unintelligible. One team chose simple batch scripts for the deployment with TeamCity managing the build and test steps, whilst another team chose Node.js simply because it was the same language they were using to develop the application.

Even though the consuming projects themselves had been split up they were still tied together by shared

[24] https://speakerdeck.com/chrisann/evolving-from-a-monolithic-to-a-distributed-public-api

libraries which held unknown, and possibly untested, quantities of business logic. Any changes to these shared libraries had to be co-ordinated between the teams to ensure that they pulled in the latest fixes.

Using TeamCity we changed the process around such that changes to the shared libraries were picked up and pushed into the consuming applications. This removed a barrier to refactoring the shared libraries - the work involved in ensuring consuming applications are updated - and so many more bug fixes and improvements were made to them. This did cause some problems where a bug would creep into the shared library and break every consuming application or when the applications were not in a position to receive changes (such as when working on a large refactoring), but we chose to receive fast feedback and consume smaller changes to the libraries than have it mount up into a large, scary change.

When the majority of the platform had been split out we turned our attention to replacing the shared libraries with services. This way we could isolate the domain they were intended to encapsulate and have the logic in one place - as per the SOA approach. With frequent deployments to the consuming applications these changes could be done gradually, first by wrapping the calls to the shared libraries then by replacing the wrapped functionality one piece at a time until the library was no longer needed. There was no big bang release where the libraries were removed, it was done in small continuous changes with little to no impact on the end consumers.

The dance of the minuet

Kanban was our chosen method for managing changes. Each team had their own kanban board, backlog and roadmap. We found that keeping our Work in Progress limit small promoted frequent releases and ensured that changes did not hang around unreleased for any length of time. We were able to experiment by implementing a change, releasing it quickly, and monitoring what happened.

Monitoring is an essential part of Continuous Delivery. If you are releasing changes in quick succession, you are doing so in order to gain feedback. We employed many tools for our monitoring including NewRelic, statsd and a logging platform comprising of Redis, Logstash, ElasticSearch and Kibana.

Our monitoring gave us information about the performance of the platform, error data and its usage. If we had a theory about a particular area that may be causing a performance issue we would add metrics around it to get a baseline before making changes and watch for any improvement. This would be done in a series of releases, facilitated by the Continuous Delivery process. With the smaller applications and focussed teams we were able to try out changes to many areas of the system in parallel.

With the replacement of existing functionality, such as a shared library providing a user lookup to an internal API call with a REST URL per User id, we'd first add metrics around the current functionality. We would add a counter for the number of calls, a counter for the number of errors, and a timer. This would give us our baseline. We would replace the user lookup code with a call to the internal API and monitor the effect this had on the metrics. If it was detrimental we would roll back the change and investigate further.

Rolling back is another essential feature of Continuous Delivery which we used often. Being able to recover quickly from a bad change allowed the platform to continue to serve requests with minimum downtime. We implemented rollback as a redeploy of the last known good state. It was as quick as a normal deploy as it used the same process and ran all the relevant smoke tests upon completion. If there was any doubt that a change had caused negative effects then we rolled it back and investigated without the added pressure of downtime in a production environment. We also had all the data our monitoring tools had collected during that bad deploy to help isolate what had caused the issue.

When serious downtime did occur we had to take steps to ensure it didn't happen again. We held blameless post-mortems to ascertain how a scenario came to be, and created actions to put in place changes to

prevent a recurrence. It is very important such discussions are blameless otherwise it becomes extremely difficult to discover what really happened and to make changes. We realised that we were all part of a system and that a series of events, rather than a single event led to the downtime, and so we need to change the system. The actions were followed up in a weekly meeting.

The closing sonata

Continuous Delivery at 7digital is more than the technical challenges. The changes made were not only to the code but also to our culture and how we approached development work.

Improvements to our automated testing meant the role of Quality Assurance moved to the front of the process rather than the traditional position of being after a release candidate has been created. Instead of verifying the accuracy of changes made, QA helped us to ensure that the changes we were making satisfied the requirements and that our understanding of the changes was correct. Together the developer and QA would devise acceptance criteria and tests, including automated acceptance tests, integration tests and unit tests.

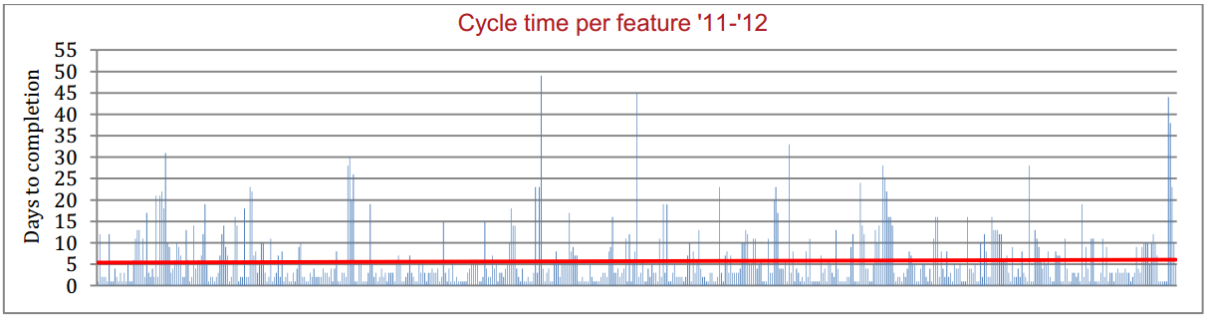
The frequent releases, rollback procedure and monitoring allowed us to spike out a change and test it in production with real live data. For example, if we believe that caching user details would be advantageous we could add a simple cache with a short timeout and monitor. If the spike proved successful we could then improve the caching strategy to add redundancy, graceful fallbacks etc. This changes the way roadmaps are devised and how closely we work with Product Managers.

The 7digital development teams no longer sit together, but rather they are situated near their internal clients - the Payments team are near the Customer Operations team, the Media Delivery Development team are near the Content Operations team, the API Routing team are near the Account Managers and so on. This promotes trust and transparency between the teams adding to greater co-operation - we took full advantage of the 'Water Cooler Effect' for incidental conversations and creating relationships across departments.

It can be appealing to be continuously deploying changes all day, but we added some rules around it to ensure a good balance - no releases after 4pm, and no releases on a Friday. This may sound counter-intuitive to the trust we have in the system, but it ensured we maintained a sustainable pace and that people were focussed when making a release. A problem caused by a bad release could take hours to manifest (e.g. a memory leak), so preventing releases after 4pm ensured that people were available to notice such issues.

The same rule applied to all of Fridays, as there are two whole days over the weekend where people may not be available. There was of course the option of agreeing a developer on-call support rota and allowing releases at any time, but this felt like an anti-solution when a sustainable pace is desired.

7digital's cycle time has demonstrably improved since these painful and laborious architecture changes were made. The work was difficult, took a very long while and at times it felt like a Sisyphean task. We pushed on through and now the API Platform is continuously being released to production as small units multiple times a day, averaging 10 or more deployments.

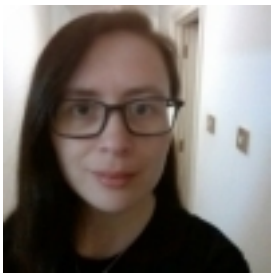


Cycle Time per feature 2011 to 2012

[25]

About the contributor

Chris O'Dell *is a Senior Developer at JUST EAT. She has nearly ten years experience working on the back-ends of web based services, primarily in .Net, most recently focussing on Web APIs. Chris has a keen interest in Test Driven Development, Continuous Delivery and Agile development practices. She lives in London and in her spare time has begun learning to play the Cello.*



Chris O'Dell

[25] http://developer.7digital.com/blog/development-team-productivity-7digital

DevOps-ifying a traditional enterprise - Niek Bartholomeus

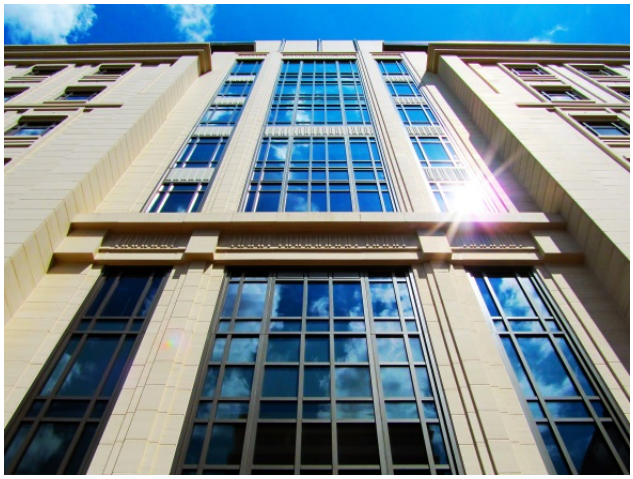
Niek on Twitter: @niekbartho[26] - Niek's blog: niek.bartholomeus.be[27]

A large investment bank in Europe

Timeline: April 2007 to August 2012

Introduction

Between 2007 and 2012 I had the chance to work in a cross-cutting team within the dev side of the IT department of a large investment bank in Europe. The objectives of this team - that throughout several internal re-organisations had been given very different names like 'Strategy & Architecture', 'Technical Architecture', and 'DevTools' - were never very clear, although they could be broadly summarised as "doing all the things that could benefit more than one development team". Nonetheless the work was very interesting and each day had its own unique twists and turns.



[28]

The team consisted of between four and eight people, who were technical experts specialised in one or two of the organisation's supported technologies (such as Java, .NET, ETL languages, reporting). I was the only true generalist in the team so work that required knowledge of multiple domains simultaneously was therefore my "specialty".

Initially we focused on creating re-usable building blocks for each of these technologies, going from defining the company's preferred application and security architectures to building framework components for security, UI templates, a common build platform, common deployment scripts, and so on.

This work - although of technical nature - had plenty of interesting cultural challenges as well, as it required finding a common ground between all of the different development flavours practiced within

[26] https://twitter.com/niekbartho

[27] http://niek.bartholomeus.be

[28] Photo by bigmacsc99 on Flickr - https://www.flickr.com/photos/bigmacsc99/4325336251 - used unmodified under a Creative Commons license: https://creativecommons.org/licenses/by-nd/2.0/legalcode

the organisation, and then convincing each team that the chosen solution is the best for the company, although it might not have been the best for that particular team.

It took a while but once this technical platform had finally settled down it proved to be of good value, not in the least for new development teams that were brought in who could hit the ground running by relying on these building blocks for all of their cross-cutting concerns.

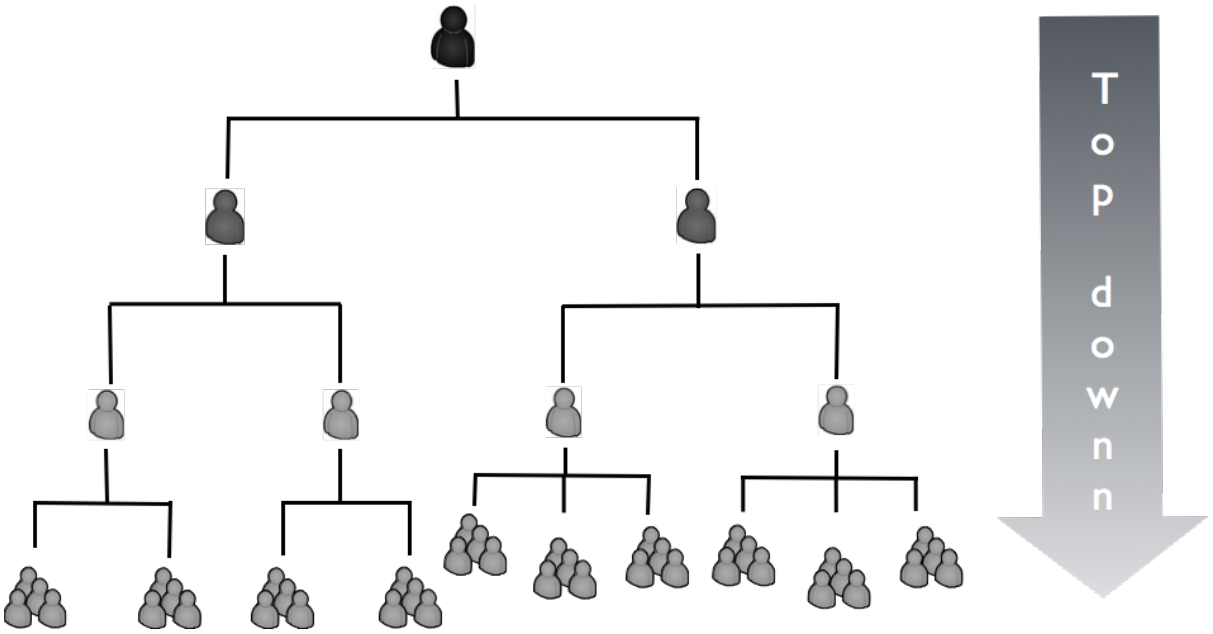
On the other hand, all that automation had not been able to contribute very much to the solution of what had by that time become the biggest bottleneck for delivering the software to the end users: the issue of the **infrequent, organisation-wide releases that remained very brittle and labour-intensive**. A different approach was needed to tame that beast, and for the generalist in me this multi-domain challenge attracted me like a magnet!

Let me first explain the organisation structure in more detail before discussing the problems that caused this bottleneck.

Organisation structure

Managers

As with so many other traditional enterprises, the company relied heavily on managers to get the work done: they cut the total work to be done in pieces by speciality (analysis, development, testing, …), assign it to their team members and coordinate the hand-overs between them. With this kind of micro-management the team sizes have to be kept small enough to avoid the manager drowning in work. This typically results in steep and hierarchical organisation structures where higher management gets separated from the work floor by several layers of middle management. As a result a huge gap is created between the place where the decisions are made (at the top) and the place where they are executed and where in many cases the deep knowledge sits (at the bottom).



A typical hierarchical organisation structure

Planning

Something else that is typical for these enterprises is their heavy reliance on planning. There is **a general assumption that the world is simple, stable, and deterministic, and therefore we can perfectly predict it**. Based on this mindset the most efficient way to execute a task is to rigorously plan up-front all the work that is needed and to assign it to specialist teams or individuals, further increasing the need for managers and coordination.

[29]

This is also where corporate process frameworks like CMMI and ITIL come in. These frameworks assume that our business is so mature that process-analysts who are far away from the reality can standardise the work we need to do into detailed procedures. This approach to structuring an organisation has some interesting consequences, which we will now explore.

Silo-isation

First of all there is the 'silo-isation' that comes with these specialist teams. People are motivated to stay inside of their domain of expertise - to 'increase the efficiency' - and leave coordination to the project managers. I have always been surprised by the little attention that generalists receive in these environments. A new problem that arises cannot always be divided up-front over the various specialist teams, but rather needs people with good understanding of the bigger picture and an 80% knowledge of multiple domains to find a good solution.

[29] Photo by U.S. Army on Flickr - https://www.flickr.com/photos/soldiersmediacenter/8405659136 - used unmodified under a Creative Commons license: https://creativecommons.org/licenses/by/2.0/legalcode

 [30]

In such a context there is also little room for experimentation. Rather the expectation is that people come with solutions by applying reductionist thinking in this supposedly deterministic world. The assumption is that we can fully predict upfront the world into hard requirements (instead of mere hypotheses) so there is no need for experimentation. If these requirements turn out to be wrong, it can only mean that we have not spent enough effort on planning so the thinking goes.

Centralisation

Secondly, the most difficult problems are typically solved in such a planning-heavy organisation by bringing in some form of centralisation. For example, if there is a big need for data to flow between applications and people get the feeling that work is being duplicated in order to combine, analyse, or transform that data then immediately this sets off a red “bad efficiency” alert throughout the management departments and significant effort is spent on rationalising the situation by adding a central data hub solution that sucks in all the source information, integrates it, and makes it available to any application that may need it.

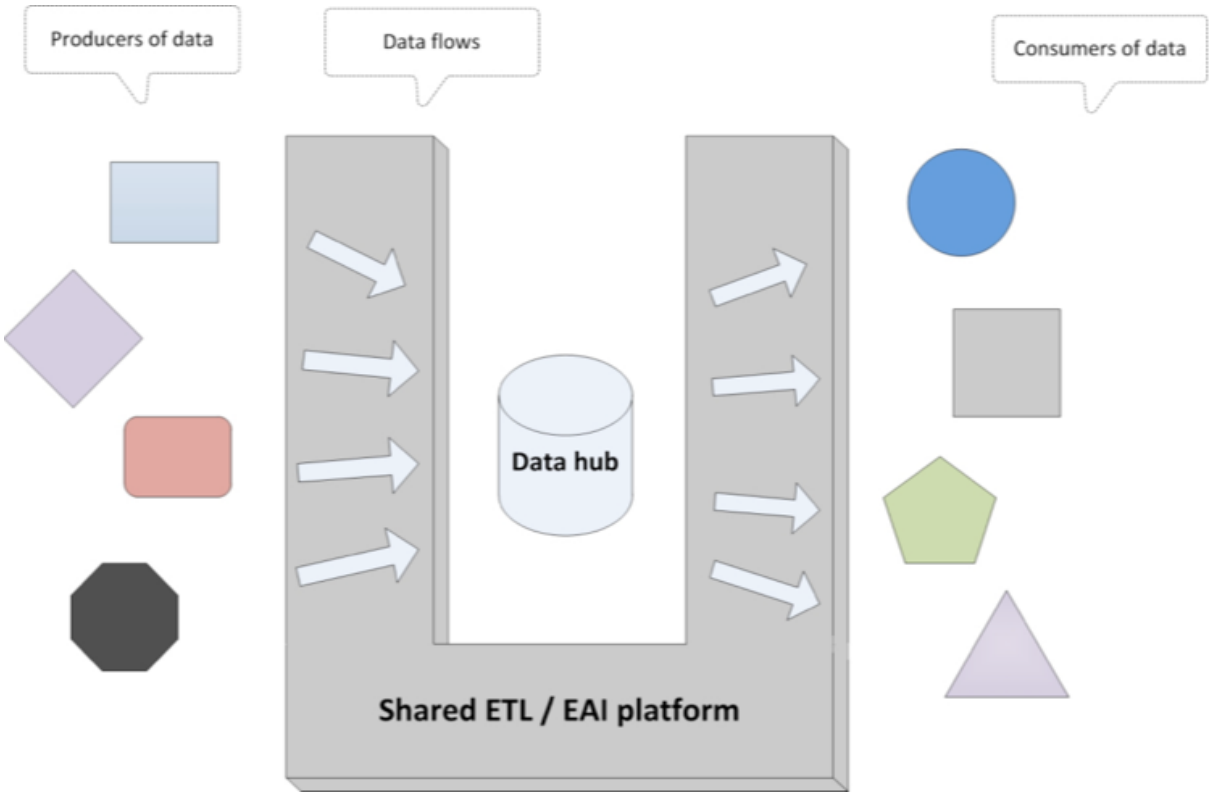
Another example concerns software delivery: as soon as the number of moving parts that has to be delivered into production reaches a certain threshold, an organisation-wide release management team is brought in to take control over the situation.

Anything for which the solution is a company-wide configuration management database (CMDB) or messaging bus are usually also good examples of this phenomenon.

Application landscape

Furthermore, the organisation was characterised by its hugely entangled and heterogeneous application landscape (in terms of technology and architecture), in which most of the applications were acquired on the market, not developed in-house. Many of these applications were tightly integrated between one another and depended on older technologies that did not lend themselves very well to automated deployment or testing.

[30] Photo by nakrnsm on Flickr - https://www.flickr.com/photos/nakrnsm/3898384586 - used unmodified under a Creative Commons license: https://creativecommons.org/licenses/by/2.0/legalcode



The application landscape - entangled and heterogeneous

Manual work

In general there was a lack of automation throughout the whole software delivery lifecycle. This in itself is quite interesting because one could argue that automation (of business processes) is what we as a department do for a living. Keeping track of which features were implemented in which versions of the application, automated acceptance testing, automated provisioning of test environments, deployment requests, release plans, all kinds of documents in order to pass architectural or project-level approvals, and much more was all done the artisanal way using Word, Excel, and a lot of manual human effort.

Infrequent, organisation-wide releases

All of the above, but especially the high trust in planning, the many (known and unknown) dependencies between the applications, and the many manual steps, led to releases that occurred infrequently and that tied together all the applications that needed upgrading, which in turn led to huge batch sizes (the amount of changes implemented in one release cycle).

[31]

Problems

An uncertain world

In addition to the problem of huge batch sizes, the whole process of software delivery had several other problems that were all rooted in one fundamental problem: the fact that it is simply impossible to predict in a sufficiently precise manner the context in which the application will exist once delivered to the end users. Even in a relatively mature domain as investment banking, there are just too many unknowns, in terms of the exact needs that the users have, the way in which all these complex technologies will behave in the real world, etc. This lack of information, this existence of uncertainty, is further increased by the high degree of silo-isation that exists. Take for example the developers: they may know all about their programming language, but they have only limited knowledge about the infrastructure on which their application depends, or about how their end users act and think exactly. They are shielded away from all these domains that may have an impact on how best to write the application code. The same applies for all other specialist teams involved in delivering or maintaining the application, each having only a partial comprehension of it.

False assumptions

Many false assumptions exist within a heavily-siloed organisation, and these assumptions will only be exposed when the application is finally deployed and used in an acceptance test or even production

[31] Photo by ajmexico on Flickr - https://www.flickr.com/photos/ajmexico/8093997590/ - used unmodified under a Creative Commons license: https://creativecommons.org/licenses/by/2.0/legalcode

environment. Operations people who interpret the deployment instructions incorrectly, developers who don't understand how operations have set up a piece of infrastructure, what the exact procedure is to request their services, etc. All of these issues take time to resolve and this unplanned time gradually puts a bigger and bigger pressure on the planning downstream. Eventually one of the deadlines will not be kept, and this will have a domino effect on all the other teams involved. In our case this resulted in testers not having enough time for regression testing (or worse: testing all of the new features), workarounds and shortcuts being implemented due to a lack of time to come up with a decent solution, new features needing to be pulled out of the release because they were not finished in time, release weekends running late, etc.

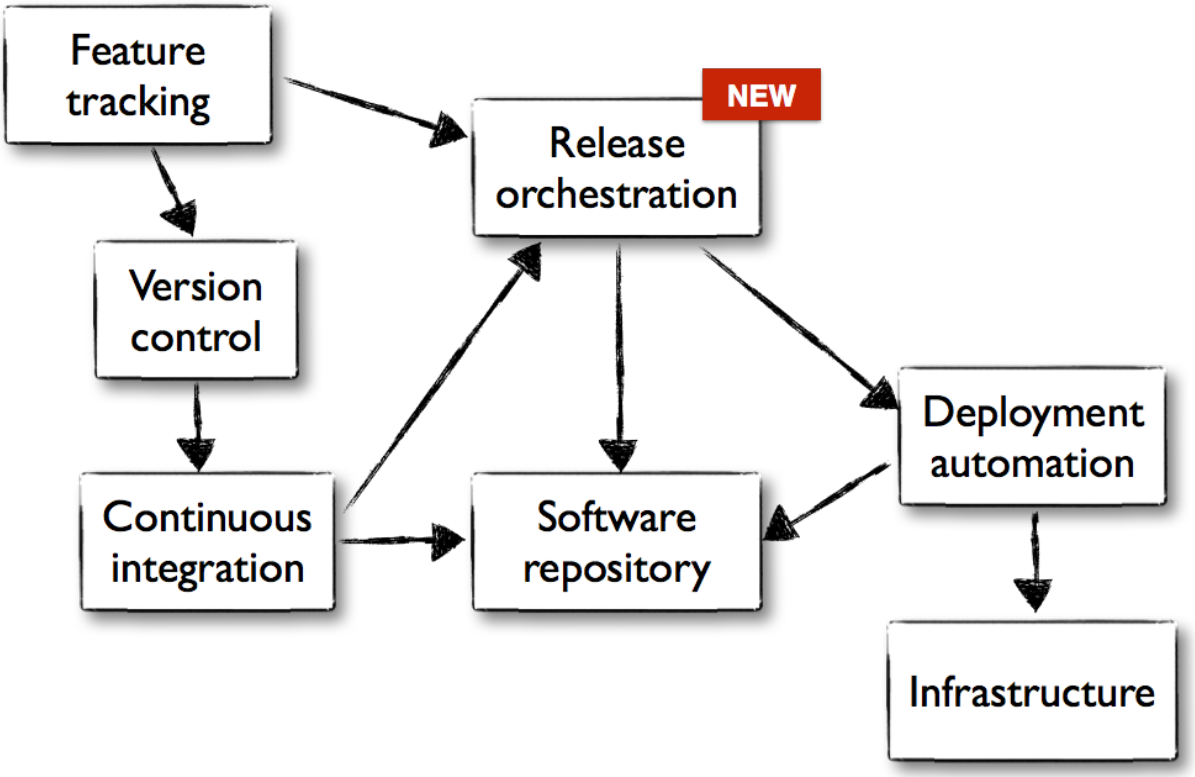
Tactical solution: enhancing the existing communication flows

I would like to say that we solved the problems by switching to a more agile approach that favours experimentation and a quick feedback cycle between idea and production that allows to spot discrepancies between assumption and reality early on. Unfortunately I only got this insight long after I left the company, when I had had the opportunity to take a step back and see things from a distance. I guess it was just too difficult to think out-of-the-box as long as I was still inside it.

Instead we focused on making the existing software delivery process more reliable by first streamlining the process level and then by automating it as much as possible.

On the process side we made sure that we came up with a process that was the simplest possible, was agreed by all stakeholders (and for software delivery that is quite a few) and was understood by everyone else involved. One of the positive consequences was that people got a better insight in what the other teams were doing which in turn led to developers and ops people starting to appreciate better what each group was doing. They finally had a common ground from which to start discussing whenever an incomprehension between them arose.

On the automation side we decided to introduce a collaboration tool to facilitate the manual work involved in tracking multi-application releases, and integrated it with our existing tools for feature tracking, continuous integration, and deployment automation in order to keep the manual work to a minimum. With the tools taking care of all the simple and recurrent tasks, it allowed the people (and the release manager in particular) finally to start focusing on more important, higher-level work. Using tooling to keep track of which versions of your application exist, which version is deployed where, how the application should be deployed, and so on avoids the human errors that would have typically caused lots of troubleshooting and stress downstream, and also increases the level of trust people put in this information.



The software delivery flow, showing Release Orchestration added

Looking back at this project two years later, I realise now that it was only the first step in solving the problem. By improving the quality of the existing communication flows we indeed considerably increased the chances of getting the releases out in time, and we definitely made the whole process more efficient, but it didn't lead in any way to an increase in the *frequency* of the releases.

See here the score card after this first step:

- Reliability: check
- Speed: uncheck

The next step should now be to shorten the release cycle, to make releasing software so easy that nothing stands in the way of doing it whenever the need occurs to validate your assumptions in the wild; that is, to finally get the quick feedback cycle that is necessary to come up with a working solution in a complex and constantly changing business.

Let me briefly explain the obstacles that still stood in the way of speeding up the release cycles and how I would now go about solving them by introducing decentralisation.

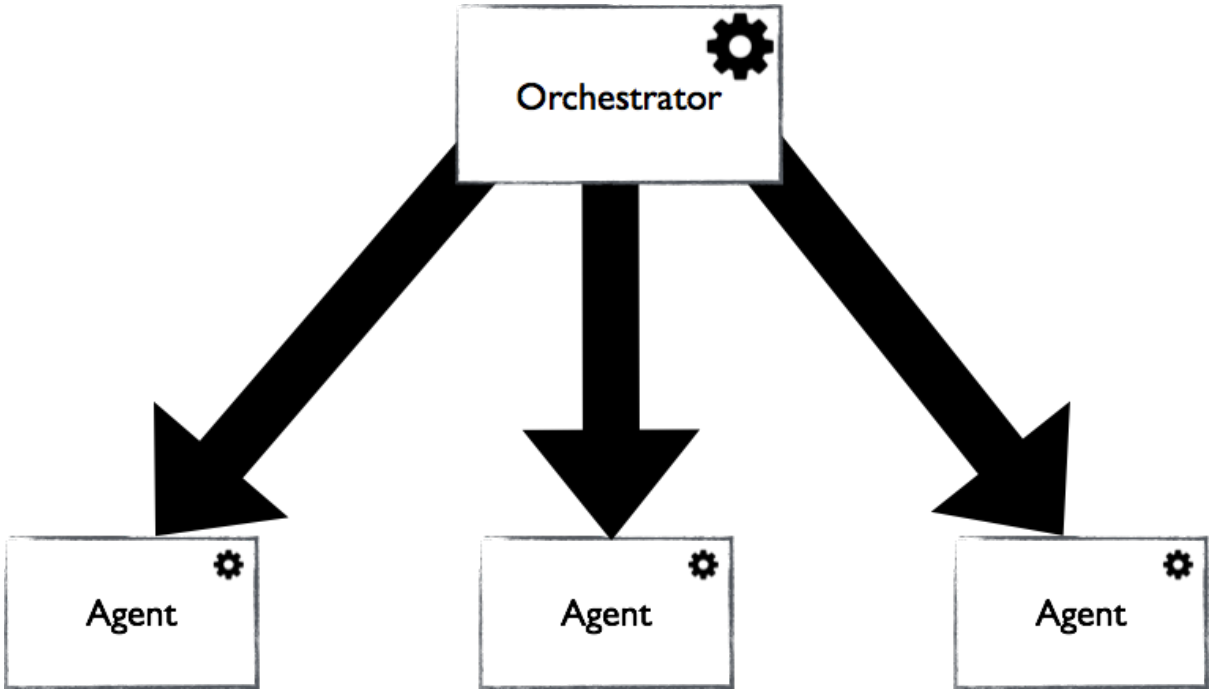
Structural solution: decentralisation

Scalability issues

The heavy reliance on centralisation that was traditionally used as a way to solve the data integration and release management problems [32] turned out to require a huge communication channel between the

[32] Even the top-down management style can be considered a result of this centralisation strategy.

central orchestrator and the agents it conducted. Therefore, as the problem domains gradually scaled out, this solution required more and more efforts to keep up. By enhancing the existing communication flows we got ourselves out of the worst mess but we could easily see that it was just a matter of time before even this solution would be pushed to its limits.

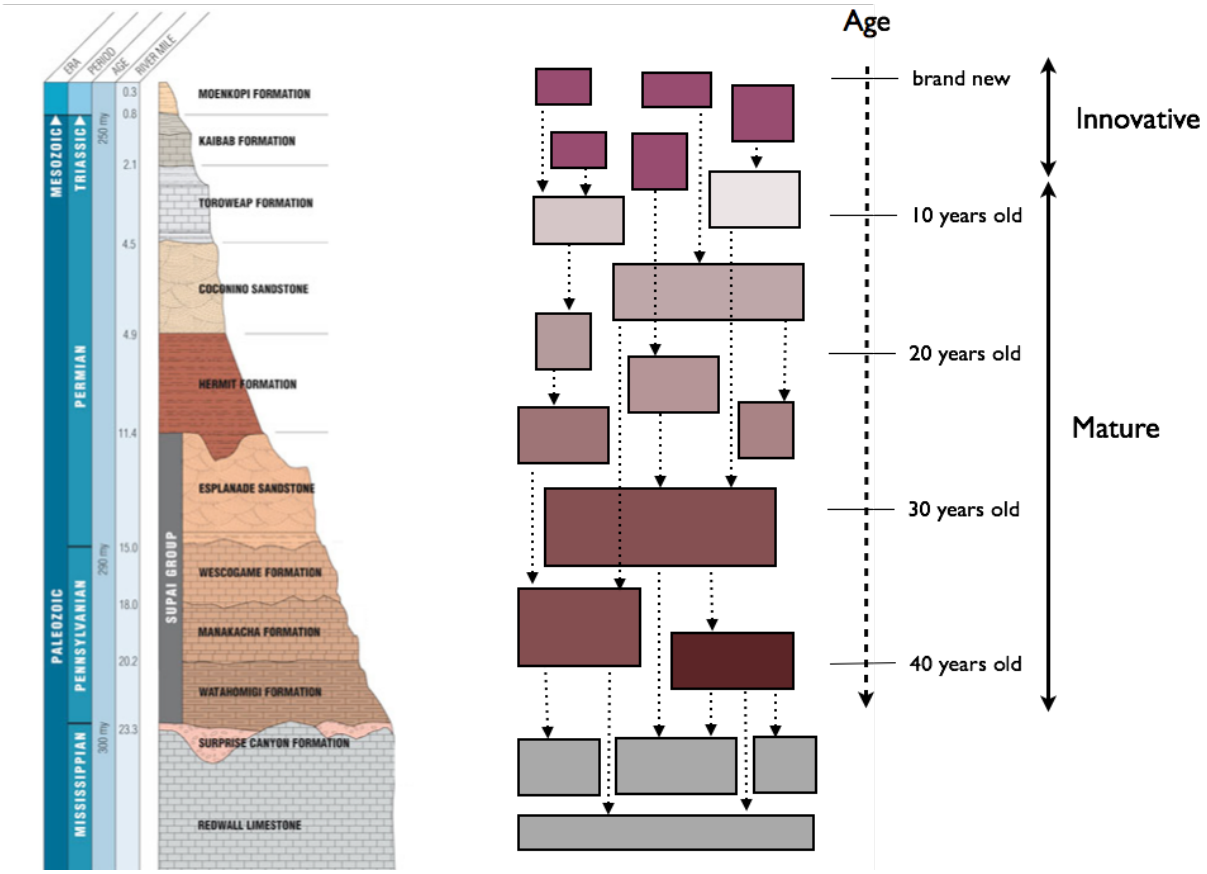


Centralisation of Orchestrator and Agents

A tendency to over-standardise

Another consequence of this centralisation was that there was a natural tendency by the central orchestrator to standardise the behaviour of its agents into a common template. The reality was that there were a lot of very different applications out there, each with their own preferred release cadence, risk profile, business maturity, technology stack, etc.

The online applications typically live in a quickly changing business and therefore demand a rapid release cycle. There are huge opportunities in these markets and risks have to be taken in order to unlock these opportunities. The back end applications on the other hand have been around a lot longer already and their market has had the time to mature, therefore it has become a little easier to make upfront predictions based on previous experiences. Also, cost-efficiency is more important here because the opportunities to create the value to cover for these costs are limited. These applications are sometimes referred to as the core applications because so many other - more recent - applications depend on it, which also makes it more difficult (in terms of total cost, risk, etc.) to change them. The drive to change them is small anyway because their business doesn't change that often anymore.

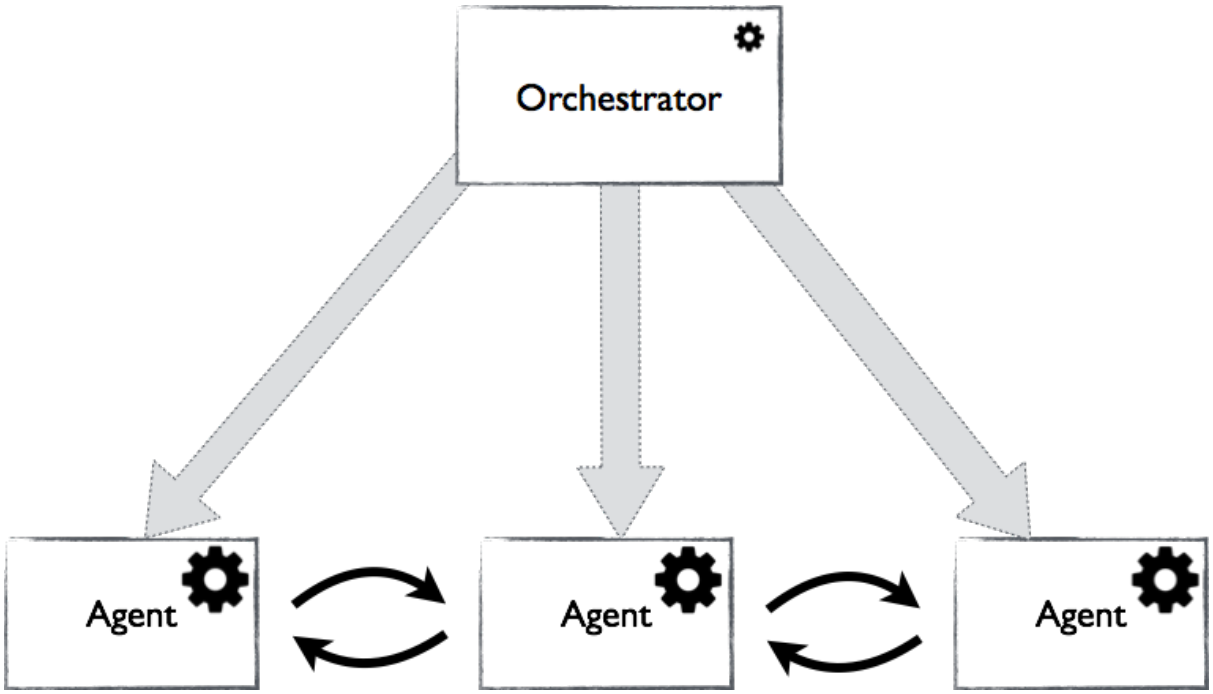


The 'palaeology' of innovative vs mature apps

As such, each individual application had a very specific profile, ranging from innovative to mature. It was obvious to me that squeezing them into a common one-size-fits-all structure had a big cost attached, although this cost was not always fully visible up front.

Decentralisation to the rescue!

To avoid these problems with scaling and standardisation, I realise now that it would be better if we could have 'loosened up' this tight coupling by pushing down the finer-grained decision-making power from the orchestrator into the agents and similarly by allowing these agents to collaborate between one another instead of always having to rely on the orchestrator for all coordination needs. If there are agents that require close interaction, it makes sense simply to bring them closer together (physically or virtually) or to combine them into one agent so the communication becomes more local and therefore more reliable. With the decision power that the agents gained they are then also free to optimise it to their own specific needs instead of having to follow the centrally imposed standards.



The decentralisation of Orchestrator and Agents

Decentralisation applied to software releases

Translated to our problem of infrequent releases this decentralisation would mean that we should first of all get rid of the application integrations that are not strictly necessary (take the use of shared infrastructure as an example) and then to decouple as much as possible the inherent integrations that remain. This decoupling can be done by making all the changes to the application backward-compatible. Yes, the magic word here is *backward-compatibility*! Make no mistake, this is a incredibly difficult task that goes to the root of how you architect and design your applications. However, once you have put the efforts to ensure backward-compatibility you will get back the freedom to release your application whenever you want, and as fast as you want, independently of all the other applications and independent of any corporate release schedules that may exist. No matter which of the other domino blocks may fall, they will not be able to touch yours. The decision power is hereby moved down from the central orchestrator - the release management team - to the individual agents - the development teams, who become autonomous and self-empowered.

 [33]

And to keep it within the spirit of autonomy and self-empowerment, in my view there is absolutely no need for all the applications to start this journey towards decentralisation at the same time and pace. The applications on the innovative side of the range would naturally benefit more from increased autonomy so it makes sense to start with them. The other applications could be done at a later time or not at all, whatever makes most sense in their specific case.

With the introduction of decentralisation the score card can hopefully be updated to:

- Reliability: check
- Speed: check

Summary

We have seen that at one point the biggest bottleneck for delivering software in the company I worked for was the fact that the releases happened infrequently and tied all applications together. We were able to trace down the reason for this to an organisation structure that relied too heavily on managers and upfront planning (resulting in heavily silo-ised teams and centralised decision making), a hugely entangled application landscape, and a high degree of manual work.

When building software for complex and quickly changing business domains it is impossible to rely so much on upfront planning because the world is simply too uncertain and there are too many false assumptions to work with. Instead we need a quick feedback loop between *idea* and *production*. This can only happen when software can be released frequently, with minimal effort.

We were able to reduce the biggest problems of these infrequent releases by improving the existing communication flows, both in terms of the process, and on the automation side. This greatly improved the reliability but didn't really do much to actually speed up the release cycles.

[33] Photo by jidanchaomian on Flickr - https://www.flickr.com/photos/10565417@N03/6246539670 - used unmodified under a Creative Commons license: https://creativecommons.org/licenses/by-sa/2.0/legalcode

The next step should now be to increase this release frequency by introducing decentralisation. Where the first step only had an impact on the process and automation side, this step will address the cultural side of the company, attempting to move the minds from a focus on determinism, upfront planning, top-down management, efficiency, etc to one with a focus on self-empowerment, mutual collaboration, experimentation, and accepting failure.

To me this looks like a crazy difficult challenge, one with no guarantee on success and with lots of pit falls underway. But still one that we should attempt, because there is not really an alternative, is there?

About the contributor

Niek Bartholomeus *is a DevOps and Continuous Delivery evangelist who has implemented a Continuous Delivery pipeline during his most recent mission at ReQtest, a small agile company. Before that he was a technical architect at a large financial institution where he was responsible for bringing together the dev and ops teams, on a cultural as well as a tooling level. He currently works as a DevOps consultant for BMC. He has a background as a software architect and developer and is fascinated by finding the big picture out of the smaller pieces.*

Niek Bartholomeus

A testing transition from yearly to weekly releases - Rob Lambert and Lyndsay Prewer

Rob on Twitter: @Rob_Lambert[34] - Rob's blog: thesocialtester.co.uk[35]

Lyndsay on Twitter: @lyndsp[36] - Lyndsay's blog: tdoks.blogspot.co.uk[37]

NewVoiceMedia[38]

Timeline: *January 2010 to July 2012*

NewVoiceMedia provides ContactWorld, a customer contact platform in the cloud. This enables organisations to connect with their customers worldwide, enabling them to deliver a personalised and unique customer experience and drive a more effective sales and marketing team. Our platform serves millions of call minutes each day, connecting in excess of 13,000 agents to their customers, in 40 countries on five continents. In our last financial year, license revenue grew at over 100 percent, outpacing the market fivefold. Over the last 18 months, we have raised $105m in VC funding.

Our DevOps Teams are located in Basingstoke, UK and Wroclaw, Poland. For more information visit www.newvoicemedia.com[39] or follow @NewVoiceMedia[40] on Twitter.

The problem

The contact-centre market changes quickly. It's a fast-paced environment with lots of competition. Back in 2010 our processes, codebase and architecture meant we were releasing annually into this market. As you can imagine the market had shifted considerably by the time we released a new version of our service to our customers.

This slow deployment cycle created problems. Our customers' needs changed in the period between asking for features and those features being delivered. The market competition was intensifying, but our competitors were also locked into slow release cycles. We saw an opportunity to move faster, shrink our own feedback loops and gain a competitive advantage.

Our vision

We started our transformation by establishing a vision; the vision was weekly releases into production. Weekly releases was a cadence that didn't seem impossible to achieve, but satisfied the desire for earlier feedback on new features.

[34] https://twitter.com/Rob_Lambert
[35] http://thesocialtester.co.uk
[36] https://twitter.com/lyndsp
[37] http://tdoks.blogspot.co.uk
[38] http://www.newvoicemedia.com
[39] http://www.newvoicemedia.com
[40] https://twitter.com/NewVoiceMedia

The vision was proposed within Development and had the enthusiastic support of our CTO, Ashley Unitt. It took further socialising and persuasion to convince other departments involved in delivery. We recruited people who shared our enthusiasm for the vision and who could help us achieve it. We planned our future in line with our vision. We made technical choices against our vision. We focused on removing obstacles that would prevent us from releasing weekly, such as manual deployments to environments, long test cycles, lack of test automation and big requirement specifications.

We had many things in place that helped us achieve this: we had the support of senior management; we had a multi-tenant platform meaning one single version of ContactWorld; we had a well-established operations team and stable cloud environments; and we had a strong will to see the business succeed.

This article focuses on the aspect of testing, and how shrinking the test cycle and moving testing to be an activity rather than a phase helped us realise our vision.

Who does the testing?

The first thing we did was to remove the notion that all testing is done by testers alone. To help achieve this cultural change, we structured R&D into cross-functional feature teams, each with a Product Owner, Scrum Master, Developers and Testers. We repeatedly emphasised to each team that although Testers do some testing, so too do Developers and Product Owners. We made the definition of acceptance criteria and the testing against it the responsibility of the whole team, instead of just Testers. We also made sure computers did lots of automated checks and empowered the business to do manual testing before releasing.

What approaches do we use?

We adopted Behaviour Driven Development and Test Driven Development approaches. We had Continuous Integration running every unit test on each code check-in. We started doing pre check-in code reviews and pair programming: both forms of testing.

Two NewVoiceMedia developers pair programming

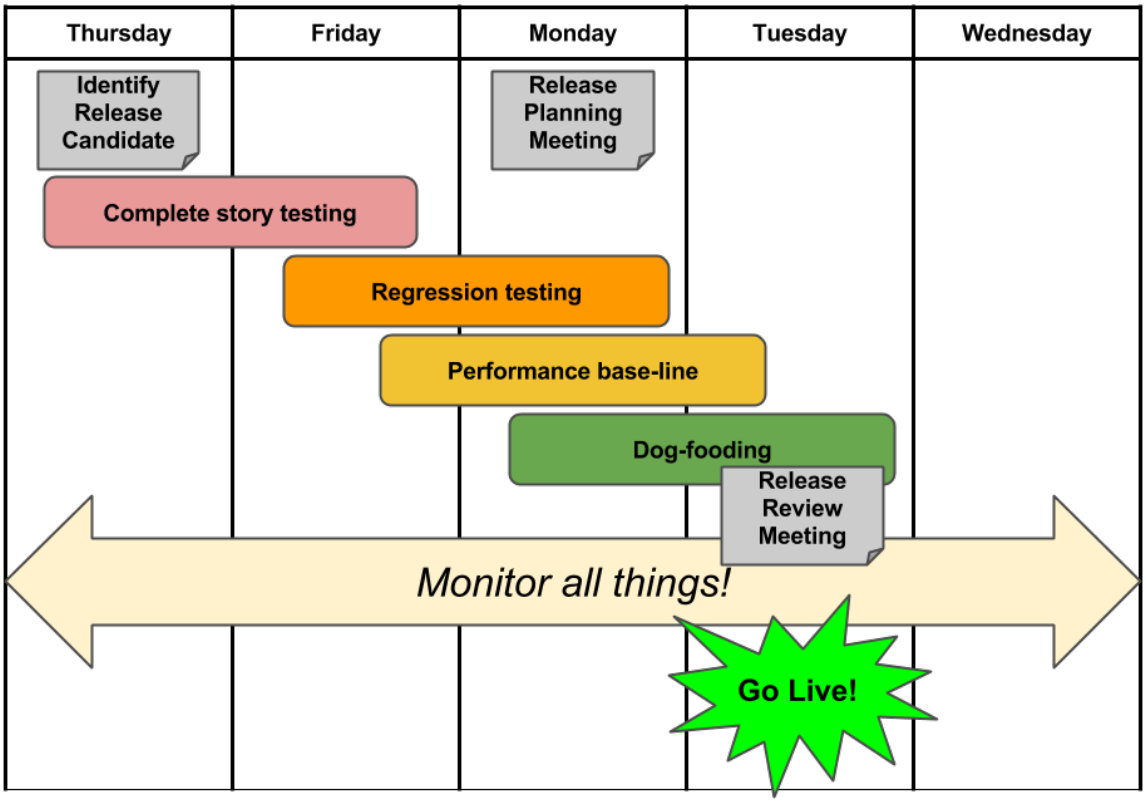
We started pairing testers and developers. We started to automate the key user scenarios using Selenium. We started doing weekly performance benchmark tests and regular security tests.

The deployment process

We moved to rapid deployment of ContactWorld's components to test environments that mimicked the production environment. We started to do exploratory testing sessions on developer environments before the code was even committed.

The whole team's focus was on establishing a weekly release candidate. This release candidate needed to have passed this battery of testing before being considered suitable for even pre-production testing.

We then moved to deploying our code to a pre-production environment for a few days before a release to production, and we started to put in place world-class monitoring on our production systems.



Our weekly release process

We also started to test ContactWorld running in the production environment from remote servers, and report the results of this to our customers on a public 'trust site'[41] . The site is still running today and gives our customers, and us, insights into the platform's stability.

Testing became the primary focus of our world. It wasn't the Testers who did the testing - everyone did.

Feedback loops

By shortening the release cycle from one year to one week, we received feedback much faster. Making testing the focal point of the process right from the start of the development cycle helped us get immediate

[41] http://www.newvoicemedia.com/support/trust/

feedback, but we also learned more about our system from production usage.

Weekly releases mean the extent of the code changes between consecutive releases is small. This small difference is a great benefit and also key to why Continuous Delivery works.

If a release causes an incident in the production environment, our rollback mechanism means we can seamlessly revert back to the previous good release. Each weekly release is much easier to analyze than the larger annual releases, enabling us to determine exactly what change caused the issue. We also gain from having production usage data that gives meaningful context to what led to the issue. Testing with model scenarios on model test environments is helpful and useful, but is no match for seeing how the service behaves under actual, live usage.

We use root cause analysis techniques, such as 5 Whys[42], with every issue we encounter, whether that be a bug in production or an operational incident. This approach allows us to get to the root cause of the problem and fix any process failures to prevent a similar issue re-occurring. Releasing weekly is not an end goal in itself; when paired with root cause analysis, it's a mechanism for providing an excellent service for our customers, by identifying and removing as many problems and issues as possible.

Another positive aspect of releasing regularly is that we can work closely with our customers to ensure that we are building the right service for them. By using Feature Toggles with small incremental changes we can Canary Release new features for individual customers, allowing us to get rapid feedback on what we are building. We can also then test, learn and improve with a small subset of users before rolling these features out to the whole user base. We are also able to switch focus quickly if the market changes or our priorities shift.

Challenges

With all dramatic change comes some pain. We had a saying early on: "If it hurts, keep doing it." This may sound strange but if something hurts enough we will always find a way around, over, under or through the problem.

One of the biggest past and present challenges for us is how to monitor the service. Monitoring is crucial to understanding what the service is doing at any given time, but also what impact we are having on the service when we deploy, patch or maintain our environments. Monitoring and associated alerting is also crucial to spotting issues before they become a problem for our customers.

The questions we face are typically how to monitor and how much monitoring to do. Since 2010 we have improved our monitoring exponentially through the use of tools that aggregate data from several sources. We have also built an impressive Network Operations Centre (NOC) where a variety of system stats and other monitoring information are visible. We've started to use tools like NewRelic, Nagios and Papertrail to give us unprecedented insight into how our platform, service and networks are behaving.

The future

One of our next steps is to instrument ContactWorld so we get deep visibility of how each component is being used and how this affects the service we provide our customers. All of this data combined with what we currently gather will give us an overwhelming set of information about our service. Our challenge then will be how to mine this data and generate new insights.

In conclusion, the market we operate in forced us to rethink how we could accelerate the delivery of our service to our customers, and this naturally led to a Continuous Delivery model. Ironically, since moving

[42] https://en.wikipedia.org/wiki/5_Whys

to more frequent releases we have seen a huge increase in the amount of testing that we are doing, despite the dramatically reduced time between building something and it being used in production. With this additional testing we've seen a significant increase in quality, and we now measure the time between coding something and finding any problems in minutes rather than months. These changes are great for those building the service, but even better for our customers.

About the contributors

Rob Lambert *is an Engineering Manager at NewVoiceMedia helping to build truly awe inspiring cloud based contact centres. He is the author of Remaining Relevant, a book about remaining relevant and employable in today's shifting industry, and the author of loads of test related content from books to magazines to newspaper articles. When Rob is not at work, writing about testing or speaking at conferences he volunteers his time for not-for-profit organizations working in the ICT4D sector. He is helping to define software testing for the modern age by challenging stereotypes and defining new way of organizing and managing testing. Through his curiosity for how things are done and his deep passion for finding new ways of doing things he is slowly bringing about change in the testing world.*

Rob Lambert

Lyndsay Prewer *is one of NewVoiceMedia's Engineering Managers (DevOps). He's passionate about helping people, teams and products become even more awesome. A former rocket-scientist, over the last two decades he's helped ten companies in two hemispheres improve their agility, build, test and deployment processes.*

Lyndsay Prewer

Delivering Continuous Delivery, continuously - Phil Wills and Simon Hildrew

Phil on Twitter: @philwills[43] - Phil's blog: blog.phil-wills.com[44]

Simon on Twitter: @sihil[45] - Simon's blog: simon.hildrew.net[46]

The Guardian[47]

Timeline: *August 2011 to August 2013*

The Guardian is one of the world's leading news sites, attracting more than 100 million unique users a month. The digital development team[48] based in London has about 150 members. The team is responsible for the public website, Android and iOS apps, as well as editorial tools, analytics, and our dating site.

The Guardian went from a fortnightly release cycle in 2011 to more than 10,000 production deployments in 2013. Why and how did we do it?

Why deploy so frequently?

Continuous Delivery is the ability to ship working software to production at any point. We believe that the only way to categorically prove software is production ready is to deploy it to production. Automated testing and pre-production environments can increase your confidence, but they can never account for everything that can possibly go wrong.

Before we moved to Continuous Delivery, releasing to production was a slow, cumbersome process that involved a round of regression testing, collating a list of all the changes and a 12 step semi-automated process performed by a team who weren't involved in making those changes. If we needed to make an urgent change or fix, it needed to be applied to multiple branches and there was nervousness around whether the deployment process itself might cause issues.

We relied on pre-production environments to try and gather feedback, but nobody takes staging environments seriously. It is unreasonable to expect people with busy day jobs to use pre-production tools in a way that even closely resembles real use. Small groups of user testers for a website are never a substitute for a global audience.

One of the most frequently quoted reasons for adopting Continuous Delivery is to deliver the value of software sooner by releasing earlier. Whilst this is a benefit, given that we had been releasing once a fortnight the incremental value of doing so wouldn't have justified the effort involved.

What has been transformative for us is the massive reduction in the amount of time to get feedback from real users. Practices like Test Driven Development and Continuous Integration can go some way

[43] https://twitter.com/philwills
[44] http://blog.phil-wills.com
[45] https://twitter.com/sihil
[46] http://simon.hildrew.net
[47] http://www.theguardian.com
[48] http://developers.theguardian.com/

to providing shorter feedback loops on whether code is behaving as expected, but what is *much* more valuable is a short feedback loop showing whether a feature is actually providing the expected value to the users of the software.

The other major benefit is reduced risk. Having a single feature released at a time makes identifying the source of a problem much quicker and the decision over whether or not to rollback much simpler. Deploying much more frequently hardens the deployment process itself. When you're deploying to production multiple times a day, there's a lot more opportunity and incentive for making it a fast, reliable and smooth process. Having a hardened deploy process also means that pushing out a fix for a bug at 5pm on a Friday afternoon becomes a low risk, routine activity.

Not everyone was convinced from the outset though. It can be counter-intuitive to suggest that releasing more often reduces risk.

How did we make it a reality?

In order to convince everyone of the value of Continuous Delivery, we rolled out the new release process slowly. We started with lower profile, less risky systems using Google App Engine which made deploying frequently much easier. Once we had this in place, then we made sure to make the most of it. We deployed frequently and sought opportunities to provide small features or fixes rapidly. This meant that users became advocates for Continuous Delivery, asking why other software couldn't be delivered in the same way.

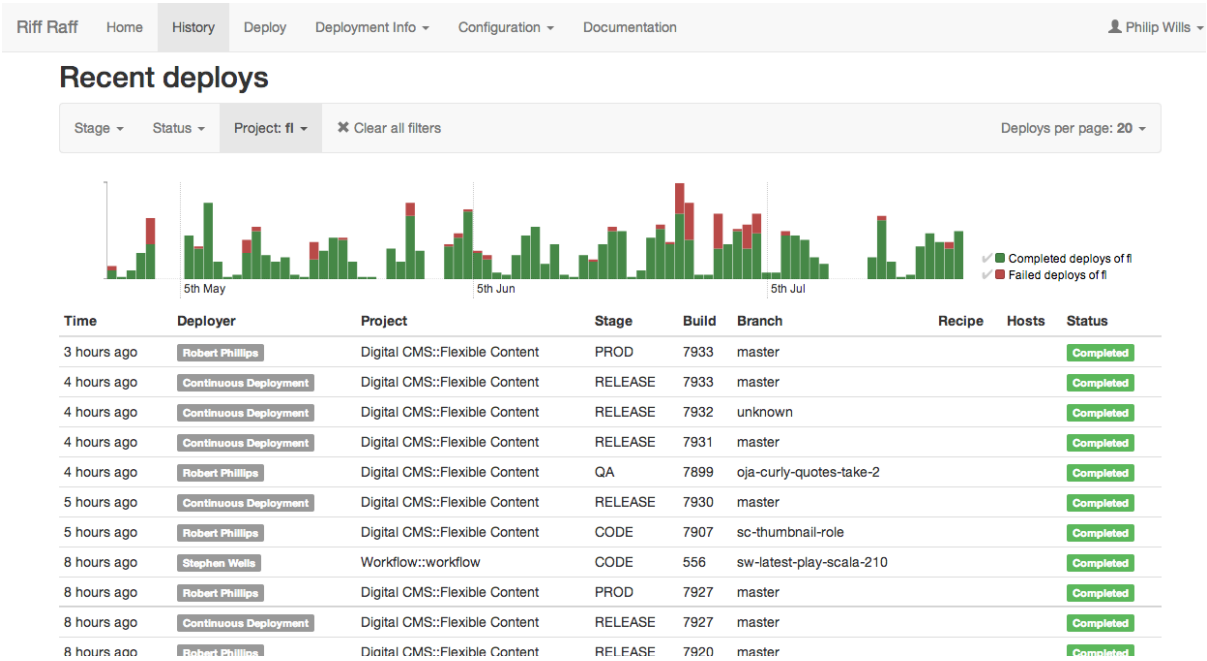
Introducing Continuous Delivery involves a carefully choreographed combination of both cultural and technical change. We knew that in order to make Continuous Delivery work for us we needed to improve the integration between our development and operations teams, and make sure they both understood and were working towards the goals of the organisation as a whole. Practically, there seemed little point moving to Continuous Delivery without breaking away from the culture of a fortnightly release cycle. In hindsight this was at odds with the success we'd had introducing Google App Engine, but we found ourselves waiting for a cultural change before implementing tooling to support Continuous Delivery. As a result the adoption of Continuous Delivery stagnated.

We eventually decided to start work on the tooling anyway. Whilst we haven't reached DevOps nirvana, once developers were deploying to production more frequently they became a lot more interested in how their code actually behaved in that environment. Tooling changes and cultural changes moved forwards in parallel and reinforced each other.

RiffRaff, a tool for simple deployment

We were already using TeamCity as an off the shelf Continuous Integration server, but we were clear that building and testing a deployable artifact wasn't something we wanted to mix with the act of actually deploying it. In order to deploy everything we wanted to, we knew some processes would be different but we wanted to limit the number of possible deploy methods so that similar systems were deployed in the same way. We couldn't find a tool which fit our needs, so we started writing one ourselves known as RiffRaff[49].

[49] https://github.com/guardian/deploy



RiffRaff deploys, including examples of Continuous Deployment and manual deploys

Again, we took an incremental approach. At first it was a command line tool, that could only deploy Java webapps within our data centres. It's now a push button webapp that can deploy a variety of different types of application to internal and cloud hosts as well as uploading files to S3, applying CloudFormation templates and pushing CDN configuration updates. It was started by members of our development team and is written in Scala, which isn't well known as a systems language. We were lucky enough to have a member of the operations team with a background in development who was keen to learn and took over leading it's direction.

This operational focus helped us to identify and eliminate a deployment antipattern. Prior to RiffRaff the operations team would carry out all deployments. A developer would raise a ticket for operations to deploy a particular version of an application into production. Operations would pick up the ticket within an hour or so, carry out the deployment according to instructions maintained on a wiki and then contact the developer so that they could test the deployment had worked. Whilst it was important that operations knew that a deploy had happened in case a deploy broke something, they didn't really add anything to the process. Operations simply held the keys and acted as gatekeeper. When RiffRaff was developed to give developers direct access to production deployments, concerns like traceability were front and centre. When things have gone wrong, identifying who changed what and when has been trivial.

One of the strengths of RiffRaff has been the flexibility provided by the integration endpoints. A comprehensive API including web-hooks allows developers to integrate RiffRaff information into dashboards, and easily start and monitor deploys from their own tool chains. So whilst we are providing a deployment platform for people to use, teams are able to use it as they see fit.

Monitoring and alerting

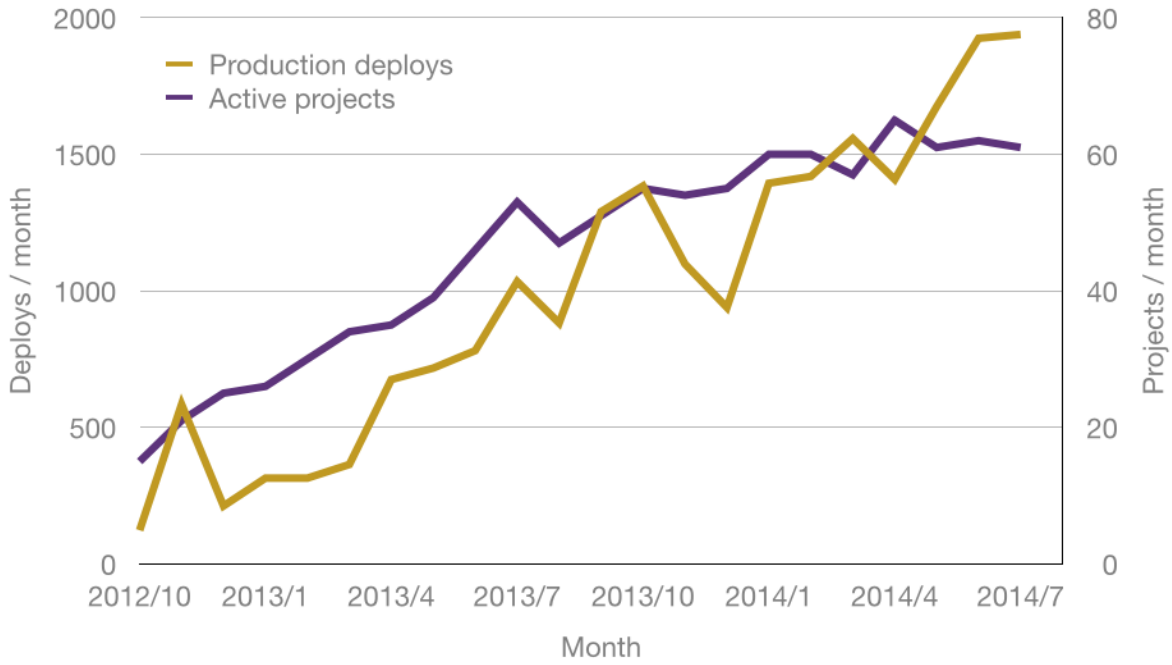
Smaller releases make it simpler to identify the cause of incidents, and having the codebase always ready to deploy makes applying fixes to problems easier. What isn't addressed directly by Continuous Delivery is the time to detect that there is an incident in the first place. To aid that we've put a different focus on monitoring and alerting. Whilst gathering vast numbers of metrics can be useful during diagnosis, for monitoring which people are actively looking at or alerting from fewer metrics are better. High level,

business-focused metrics have produced fewer false positives, whilst capturing a wider range of issues than other approaches we've tried.

We view an application with a long uptime as a risk. It can be a sign that there's fear to deploy it, or that a backlog of changes is building up, leading to a more risky release. Even for systems that are not being actively developed on, there's value in deploying with some regularity to make sure we still have confidence in the process. One note of caution: deploying so frequently can mask resource leaks. We once had a service fail over a bank holiday weekend, as it had never previously run for three days without being restarted by a deploy!

Why not Continuous Deployment?

Continuous Delivery often gets confused with Continuous Deployment, which is the practice of releasing to production automatically on each commit. Given how often we deploy it would be reasonable to ask why we don't go the whole hog and move to Continuous Deployment. While RiffRaff is used by many projects for Continuous Deployment in pre-production environments, only a few use it to push automatically into production. The automated checks to validate a deploy are not mature enough to rely on, and we found that when people didn't manually initiate a deploy they were less likely to monitor metrics during and after it.



RiffRaff deployment history

As an example, we experimented with Continuously Deploying some components of our new website, including a component responsible for rendering all of the football pages. To push out a new feature to production, a developer could simply merge their feature branch to master and rely on the Continuous Deployment pipeline. Some time after this was put in place part of the pipeline broke - whilst it appeared to be working, new releases were not actually being pushed into production. Developers continued to work on the assumption their changes were available to end users and it took two weeks before anyone noticed.

Roles and responsibilities

With Continuous Delivery tooling in place, the culture of project teams and the roles of team members began to shift. We've noticed changes in three roles:

- Developers are doing operations and support as products shift to full stack ownership
- QA have shifted from manual regression testing to automated tests and exploratory testing
- Operations have shifted from running products and being production gatekeepers to being consultants and watchmen (auditing product implementations)
- Release management is no longer a dedicated role, with each product team able to release when they like and RiffRaff providing centralised reporting

Adopting Continuous Delivery has not been simple, quick, or cheap. We've spent at least a year's worth of a developer's time on deployment tools alone. We believe this cost has been more than justified by massively reducing regression testing cycles and eliminating the dedicated release manager role, even before taking account of the less tangible benefits of a faster feedback cycle. Continuous Delivery has made us more resilient, allowed us to build more valuable software, and also made delivering software more fun. Hearing a suggestion from a user, delivering some approximation of it an hour later, and then iterating with them to make the most of it is a hugely rewarding way of producing software.

About the contributors

Phil Wills *is senior software architect at the Guardian. He has worked on just about every part of the Guardian's site and the tools which support it; building features, optimising performance and increasing resilience. Over the past two years, he's helped the Guardian achieve much more frequent delivery.*

Phil Wills

Simon Hildrew *led the Guardian's digital operations team for two years before changing role and becoming a developer (albeit with an operations focus). Since then he has tackled pain points encountered by operations, including writing tooling that enables deployments to be carried out more reliably.*

Simon Hildrew

Making the world a better place - Marc Cluet

Marc on Twitter: @lynxman[50] - Marc's blog: devroot.org[51]

Various companies, from startups to large enterprises

Timeline: *2009 to 2014*

Context

One of my main purposes at work during the last five years has been to spread and help companies embrace a DevOps culture[52]. Often this was as a consultant in smaller companies where the culture was already built and prepared for disruption (which makes things easier), but most recently I was responsible for introducing and nurturing a DevOps culture within larger, more complex organisations (banks, media giants, big software companies) which was a whole new challenge. I will describe the different challenges faced by companies of different sizes when they begin to embrace DevOps.

High level DevOps

DevOps is an evolution of our software industry in order to make projects *more successful* (this is clearly a very broad statement but fortunately there is data to back this up). In the Chaos Manifesto[53] it is affirmed that DevOps projects have a higher level of success and a lower level of challenges, which from my experience is true: DevOps projects embrace the path of least resistance while at the same time having just enough structure surrounding them to make sure that everything is measured and accounted for.

A good friend of mine (with a heavy Project Management background) said that DevOps is all about the **PPT**, which stands for People, Processes, and Tools[54]. Any good working culture (as DevOps is) should be able to cover all of those with a high degree of success and still embrace the path of least resistance[55] to make things as natural as possible. Any kind of change to any of these three factors will act like a rubber band and add tension to the other two, you try to advance Tools only and that will create tension with Processes and People and so on. This is one of the main mistakes that organisations make when embracing DevOps, as it is widely thought that DevOps is all about the Tools[56] when in reality it is heavily based on Culture (which embraces People and Processes); the Tools are just a means to an end[57] (improvement, better software systems, happier customers) and are the catalyst of those Process and People changes.

[50] https://twitter.com/lynxman
[51] https://devroot.org/
[52] http://www.getchef.com/blog/2010/07/16/what-devops-means-to-me/
[53] http://www.versionone.com/assets/img/files/CHAOSManifesto2012.pdf
[54] http://www.cmcrossroads.com/article/people-processes-and-tools-three-pillars-software-development
[55] http://en.wikipedia.org/wiki/Path_of_least_resistance
[56] https://www.ibm.com/developerworks/community/blogs/invisiblethread/entry/rational_application_developer_and_devops?lang=en
[57] http://contino.co.uk/devops-organisational-change-challenge/

Is it all about 'the Cloud'?

The main driver for many companies to embrace change for DevOps is the evergreen promise of 'the Cloud': Converting CapEx (Capital Expenses) to OpEx (Operational Expenses) and only pay for what you use in an utility model pricing. Many organisations have invested heavily in the past in their own server infrastructure and even in their own datacenters, and most of them have discovered that these added huge capital costs[58] are not in line with the primary line of business.

However, having adopted Cloud technologies, most organisations realise with time that so-called Public Cloud is more expensive in the long term, hence the huge price reductions for Public Cloud seen in 2014[59] by Amazon Web Services (AWS), Google Computing Engine, and Windows Azure (apart from the natural competition to embrace the market). Running cloud server instances 24x7 is usually more costly than renting server infrastructure[60] (the model that Rackspace and other hosting providers have) and it is not a good fit for all the workloads that you might need, as shown by all the tutorials trying to run Hadoop or Cassandra in the cloud (it is a risky business). Public Cloud providers will sell you on-demand computing and help you cut dramatically the cost and time for provisioning new servers, which is extremely useful for peaky traffic sites as news agencies and ecommerce sites during sales. The emerging trend of "Own the base, rent the spike"[61] is becoming more and more popular as organisations grow, as it is an economically savvy way to be able to cope with all your incoming traffic while at the same time keeping your operational costs to the minimum possible. DevOps fits incredibly well with this economic model; being able to embrace the maxims of DevOps will help you be prepared for that kind of infrastructure modularity.

There are some companies such as Etsy that have never invested heavily in Cloud or Virtualisation[62] and run their entire production infrastructure directly on physical hardware, and yet are leaders in DevOps practices. Other companies such as Netflix have very different models of 'peaky' traffic[63], and have found that having everything in Public Cloud is a huge 'cost buster'. In short, embracing the Cloud should not immediately lead either to renting hardware or discarding your own; the truth lies inbetween.

Introducing DevOps

Culturally, IT embraces change at different speeds and rates; this means that for people that live at the cutting edge of technology like myself, more than 95% of the organisations we meet are living in some version of the past, even the ones that come to all the DevOps meetups and have all the cool stickers on their laptops. Organisations with varying internal structures have different histories and different needs to embrace change, but nonetheless it seems that most organisations follow broadly the same steps to get up to speed with DevOps. I call this the *Chain of Causality*, which is tightly integrated around the PPT model (People, Processes and Tools).

The Chain of Causality

The Chain of Causality is in effect the order in which most organisations end up changing things to enable and support a DevOps model. There have been some times when I have suggested these steps down the chain in a different order due to the product or current organisation of the company. It is important not

[58] http://www.storagecraft.com/blog/data-centers-costs-ownership/
[59] http://www.computerworld.com/s/article/9247216/Price_war_Amazon_cuts_cloud_costs_to_counter_Google
[60] http://www.forbes.com/sites/quickerbettertech/2013/04/29/do-you-replace-your-server-or-go-to-the-cloud-the-answer-may-surprise-you/
[61] http://www.rackspace.com/blog/10-reasons-why-a-hybrid-cloud-is-better/
[62] http://codeascraft.com/2012/08/31/what-hardware-powers-etsy-com/
[63] http://www.zdnet.com/the-biggest-cloud-app-of-all-netflix-7000014298/

only to be able to help organisations to evolve and change for their own benefit but also to undertake that change in the right context: I always made sure about this when analysing how to approach this transformation scenario.

1. Automation

Automation is the first and important step towards a DevOps transformation, and is important for several different reasons:

- Reproduce the same conditions across your production platform
- Quickly send change to a group of servers
- Be able to create 'very close to production' development environments quickly

Most of the next steps in the chain depend heavily on automation, which makes this a first natural step and the one with the most immediate 'bang'.

2. Provisioning

In this context we will consider that provisoning is any action that will create a new computing resource that did not exist previously under our control. Once we have most of our basic automation 'roles' it will feel very logical to start creating and destroying instances (or physical servers), especially for development environments; the added flexibility to do so will help accelerate development. One very popular way to do so is to use tools like Vagrant[64] to be able to create working images that can be used in any modern desktop or laptop computer.

3. Deployment

Deploying an application is always a stressful situation[65], and making deployments quicker and safer has always been something difficult especially with servers that needed to be maintained or upgraded at the same time. Thanks to Automation and Provisioning, Deployment would be the third natural step in this; we can easily automate provisioning in conjunction with Automation tools, and even create new servers with the latest version of our software. I have found that automating deployment is normally the second most time consuming task in a DevOps transformation; on the other hand, this process will also open the doors to better communication between dev and ops teams which is part of the cultural transformation that we will talk about in the last point of the chain.

4. Continuous Integration Testing

When the bridges of trust between dev and ops have started to be created it is time to start integrating the DevOps culture into the development, one of my personal favourite things is to introduce Test Driven Development[66] (TDD) to an organisation in combination with Continuous Integration (CI), as the results show very rapidly. A good CI framework will help not only make developers more productive but also development iterations faster and higher in quality. In my experience this step requires a very close coordination with the development team and a lot of the 'human touch' to make certain that things are always clear and agreed by all parties.

[64] http://www.vagrantup.com
[65] http://blog.algonquinstudios.com/2014/05/05/avoid-deployment-disaster/
[66] http://www.agiledata.org/essays/tdd.html

5. Continuous Delivery for bug releases

Continuous Delivery and deployment to production of bug releases is one of the key business advantages integrated in DevOps: being able to be responsive to users about mistakes and bugs in the platform is a healthy part of keeping the competitive edge. I always found that organisations are challenged by this the first weeks they do Continuous Delivery as it feels "unnatural" to let go of the deployment steering wheel. Making sure that the organisation feels comfortable about this and doing it in different phases (maybe starting by a piece of code that is not key to the organisation) will make sure that the trust is built in the process.

6. Cultural change

When all the other pieces of the chain are in place Cultural Change is already happening: each piece of the chain will bring with it a small bit of DevOps Culture, methodology, and communication. I always left culture as the last part to consolidate, because the culture needs to be adapted and 'made to measure' for every organisation at the beginning. Once an organisation has been running in a DevOps way for several months I always prefer to go back and reevaluate the culture to see where it needs tweaking; as with any methodology and culture associated with it continuous improvement is important to keep things healthy.

Working with startups

In my experience startups are the easiest organisations to help transform, as it is not really a transformation *per se*, but more of a 'getting up to speed' task; many startups will be already doing DevOps in one way or another. The startups that I have met are at most some months in the past, due to the challenge of being operational while changing or upgrading a key piece of technology. Still, it is very important to be able to embrace that change as startups feed on the ability to keep innovating; trying new technologies needs to always be 'in their DNA', which might lead to refactoring of chunks of code or services, or being able to pivot their business model. One of the most difficult things with startups is trying to beat the emerging status quo; there are people more used to change than others and that is important to keep in mind. Also, it can be difficult for a startup to dump its business model, even when that's necessary for survival.

Embracing small businesses

A small business that wants to embrace DevOps is a difficult cookie. There are not two small business that I have found that have the same constraints or are ready to put in the required effort to go all the way. One of the main limitations that I have found is the lack of resource - both human and capital - within small businesses. It does not help that small businesses will only embrace change when they are 'looking at the abyss'; they will resist changing their ways while they generate revenue until it is slightly too late or the competition is getting very though. This also becomes worse due to the lack of knowledge about new technologies and options. The Engineers that I have seen that break this mould do so in their own personal time. Small business will be very flexible to embrace change once they have set up their mind on it, but they will need a lot of external help, that is where contractors or training can definitely help.

Help enable medium businesses

Medium businesses are very fun to work with, at least in my experience! They are normally two to three years behind the leading organisations as they are not big enough to get everyone up to speed together.

Also, the pace at which they can implement organisational change is slightly slower than a small business, so that is one of the tricky parts to address. A medium sized business will transform itself in order to give it the edge against the competition, but will not stay that way for long if it is not able to compete properly. Due to having more resources than small businesses they are able to refactor chunks of their code to embrace the latest and greatest technologies and techniques. The main DevOps blocker in this kind of business is normally that mid-level management do not know how to embrace the new culture, allowing and enabling Engineering teams to try new things and learn.

Accepting the challenge of corporations

Corporations measure their survival capability in risk taking. Risk is one of the things that any big business will analyse and try to avoid. Corporations are therefore by far the most challenging organisations to transform. Being up to speed with new technologies and embracing change is very risk prone so most corporations are set in the past in a safe three to five years to be able to run well tested code with bugs that have been documented and patched; the same applies to the software they generate (unless they are an Engineer driven organisation). On top of that there are also the capital investments already made by the company which have an amortisation time[67] and need to be used up to the time they are expected to (at least in countries such as USA, UK, and India). The corporations that I have interacted with are (in 2014) only just starting to embrace automation as the new futuristic thing; corporations expect this automation to come with commercial support so they can scale their issues quickly and push the risk to the providing company, which means that adding other changes to the mix have been so far difficult to justify. The best method that I have found to work with corporations is to create a 'testing' or 'prototyping' team which will use new methodologies with my support to be able to fully embrace DevOps; once this has been proven as a success, other small teams are able to adopt the same methods. I have not personally seen any big corporation completely transformed using DevOps, and it is a scenario that I find very difficult to see in the next years as DevOps goes against the risk-adverse way of doing business favoured by most corporations.

Teaching an old dog new tricks

Helping an organisation embrace the DevOps cultural change is no easy feat, and I have been personally burned by trying to do so in organisations that were definitely not ready for it. Before embracing a DevOps cultural transformation you should ask yourself the following questions.

- Will this improve the competitiveness of the organisation?
- Will this reduce operational and capital costs?
- Do we have the right Engineers to do this?
- Is there backing by all the C-level including the CTO/CIO, the CEO, and the CFO?
- What parts of my work culture am sacrificing for this change and do I need to compensate for them?

Especially important is the amount of pain and risk that an organisation is ready to take in order to help this change, as this will dictate not only the speed of change but also what kind of liability and risk this produce.

Typical Organisations mapped to the risk/pain quadrant

[67] http://www.nysscpa.org/cpajournal/2008/508/essentials/p46.htm

Some examples of the pain and risk 'profiles' for different kinds of organisations are:

- Financial organisations (very risk averse, low pain threshold)
- Media organisations (risk prone but low pain threshold)
- Marketing organisations (risk prone and high pain threshold)
- Software organisations (risk averse, high pain threshold)

Any transformation will only be successful if the amount of pain generated by it is less of what an organisation can tolerate (as any transformation will have pain), the measurements in the previous graph should be a guidance towards that. The amount of risk an organisation will embrace will help to speed up the transformation process while the pain inflicted is tolerable.

People management

One of the most difficult things that any cultural transformation will face is the current status quo and people's fears and worries. People in general are very change averse, but as Engineers we often do not take this into account; this is the main reason why a DevOps transformation might fail (as I discovered several times!). I was exposed to a lot of challenges from different groups due to fears of becoming obsolete, lack of expertise to embrace the new technologies and changing culture. This resistance can even drive a rejection of any effort to change. Unfortunately, this can evetually lead to an organisation going out of business: as W. Edwards Deming said "You Don't Need to Change. Survival is Optional"[68]. Preparations to face these challenges to DevOps are what occupy most of my time at the beginning of any transformation. In several organisations, I have been bombarded with all kind of complaints such as "this means I have no work anymore" and "my manager does not agree with this change", which makes it critical to have the backing of the C-level and upper management, who can be used time and again to push change forward. Winning hearts and minds is 50% of the transformation challenge, there is a very good guide that I follow from Jesse Robbins[69] which gives a very good checklist to prepare for this.

In Conclusion

Due to the pressures of the market, organisations will have to 'transform or die', as has happened all through our recent history. Who does not remember the demise of Kodak[70] or the UK motor industry[71], victims of what happens when organisations do not embrace change. Embracing DevOps is the way to be able to be competitive in the future. In 2014 most organisations are just waking up to this reality and trying quickly to get themselves on the DevOps wagon: to all of them I say "Welcome to the game!".

About the contributor

Marc Cluet *is an Engineer currently managing Operations at a Startup. A Systems Engineer, Network Engineer, Database Administrator, Project Manager and Team Manager with 17 years of experience across the globe (Spain, Switzerland, UK, US, Canada), expert in Software lifecycle and applying DevOps Cultural methodologies to all levels of Engineering.*

Marc has founded and worked at several startups and also at prestigious companies like Canonical or

[68] http://en.wikiquote.org/wiki/W._Edwards_Deming
[69] http://devops.com/blogs/devops-culture-hacks/
[70] http://www.forbes.com/sites/chunkamui/2012/01/18/how-kodak-failed/
[71] http://www.telegraph.co.uk/finance/newsbysector/industry/8470355/The-fall-and-rise-of-the-British-car-industry-timeline.html

Rackspace, he also has contributed to Puppet, mcollective, Juju, Ubuntu and has helped architect Ubuntu MAAS. He has also presented at some of the most prestigious conferences and evangelised about DevOps Culture and Systems methodologies.

Marc Cluet

Staying Xtremely Unruly through Growth - Alex Wilson and Benji Weber

Alex on Twitter[72] - Alex's blog[73]

Benji on Twitter[74] - Benji's blog[75]

Unruly Media[76]

Timeline: November 2009 onwards

Marketing technology company Unruly is the leading global platform for social video marketing and works with top brands and their agencies to get their videos watched, tracked and shared across the Open Web. Our programmatic video platform reaches and engages the half of the global Internet population who are watching and sharing videos outside of YouTube.

Founded in 2006, Unruly has 12 offices and employs 150 people globally. In 2012, Unruly secured a $25 million Series A investment led by Amadeus, Van den Ende & Deitmers and Business Growth Fund.

Unruly currently has a tech team of 25 spread across 3 development teams. Unruly makes use of Extreme Programming (XP) principles across all of these teams and throughout the rest of the business.

Unruly's product development teams are cross-skilled, with developers responsible not just for development, but future product direction and keeping systems operational as well. All production code is written in pairs using Test-Driven Development.

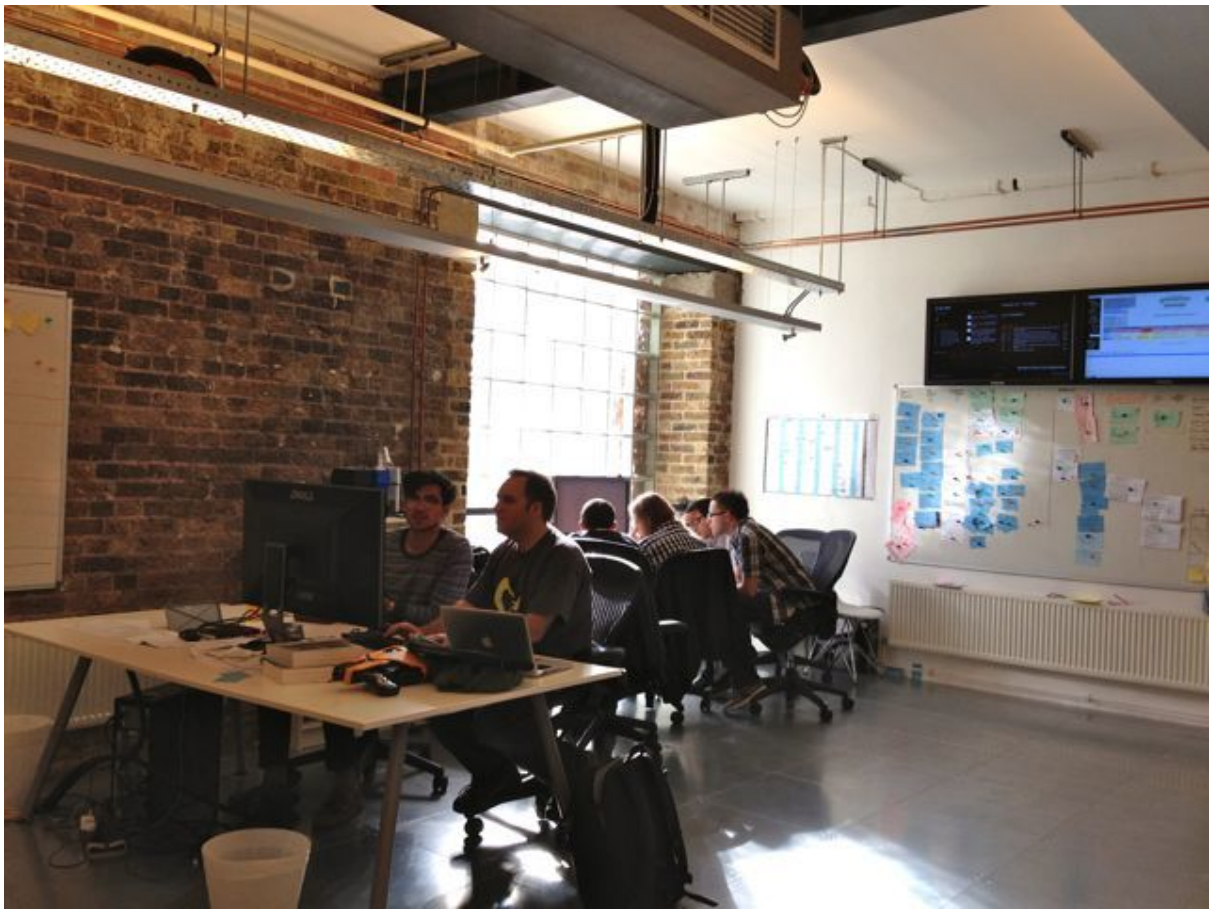
Introduction

> "I will never forget my first day at Unruly. We chose who would pair-program with whom in our morning team huddle, then sat down and wrote a bit of code. About an hour later we had made a few tweaks to the appearance of one of our ad units and the developer I was pairing with turned to me and said - 'Ok, let's deploy'. I was surprised and somewhat concerned. It was my first day and I was about to deploy changes to production that only my partner and I had tested. Where was the QA team to throw the changes over the fence to? Where was the release board requiring triplicate sign-off on changes?" — Benji

The above reaction is often echoed by developers joining Unruly. Once you get over the initial apprehension, working in this way soon becomes addictive as you can see your work generating value immediately. Each pair does a bit of work then hits the deploy button, waits a couple of minutes for the deploy script to run all the tests and release the changes. Once it is complete we can then wander over to an actual user's desk to find out if it is working as they desire.

We realised some time ago this was a fundamentally better way of working. There is no room for hours or days of wasted work building things that nobody wanted in the first place.

[72] https://www.twitter.com/pr0bablyfine
[73] http://www.probablyfine.co.uk
[74] https://www.twitter.com/benjiweber
[75] http://www.benjiweber.co.uk
[76] http://www.unrulymedia.com



Pairing in action at Unruly

Unruly has worked this way from the outset as its founders had experienced the many advantages of Continuous Delivery prior to starting the company, so we can't tell you about how much better things got after switching to a Continuous Delivery approach Rather, we've made lots of small incremental improvements as we have grown, while striving to maintain the rapid pace of delivery that we value. We'd like to share some lessons we've learnt along the way.

Testing in Production

When the organisation was small enough to fit into a single room it was easy to only deploy changes that customers were happy with. Our team's direct customers were typically other Unrulies, depending on the impact of the change we'd either get them to look at what we were about to deploy, or deploy it first and then get feedback. We could deploy frequently and receive rapid synchronous feedback on our changes.

However, as we grew and launched offices all around the globe this became more difficult. Our development team is based in London, and when we built features for people in our American offices (San Francisco, in particular) we only had a few minutes of overlap time each day to discuss our change. It was hard to find time to review changes with customers before deploying them, and consequently our deploy frequency dropped.

We considered introducing a staging environment where we could demonstrate changes to our overseas users before release, but ultimately decided this was the wrong path. It would require maintaining and

securing a lot more infrastructure, and would have required us to maintain code branches while we were waiting for feedback.

Instead, we opted to use Feature Toggles in our production environment to give selected users access to new features that were partially complete. This allowed us to continue integrating our changes with the rest of the development team and release them to production while we awaited feedback.

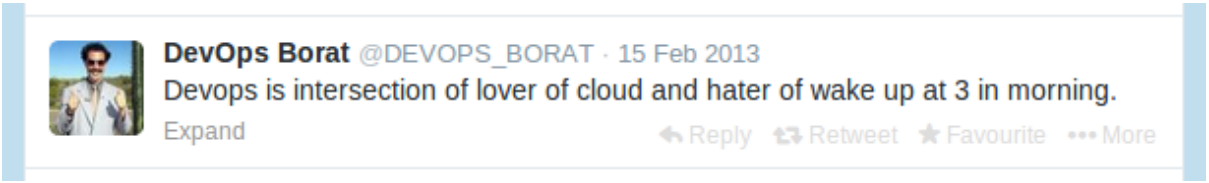
The result was that our end-of-day calls consisted of walking the customers through the features in the environment they were already using every day. We could discuss potential improvements for the next day and they could see how the new features interacted with the data and events in the live environment.

Co-ordinating Shared Infrastructure

As the rest of the company grew, so did the product development team. Eventually we found the team was large enough that we were spreading ourselves too thin and becoming unfocussed on product direction, so we split along product boundaries into 3 sub-teams - one for ad-unit management in our video distribution product Unruly Activate, one for campaign management in Unruly Activate, and one for our Unruly Analytics product. Each team had its own infrastructure concerns but everyone was responsible for shared infrastructure such as monitoring, provisioning, and configuration management.

Since we now had 3 teams working by-and-large on entirely different projects which were adding new features and systems faster than before, the parts of our set-up which were used by all teams started to become seen as "not as important", "not worth prioritising", and worst of all - "not our problem". This led to some incidents where problems with our Nagios and Puppetmasters caused production issues. The incidents were ultimately caused by defects that each team had assumed another would fix.

We counteracted this by introducing a cross-team squad responsible for these priorities, consisting of one representative from each of the teams and our site reliability specialist. We affectionately refer to this group as Borat Squad, after the @DEVOPS_BORAT Twitter account.



[77]

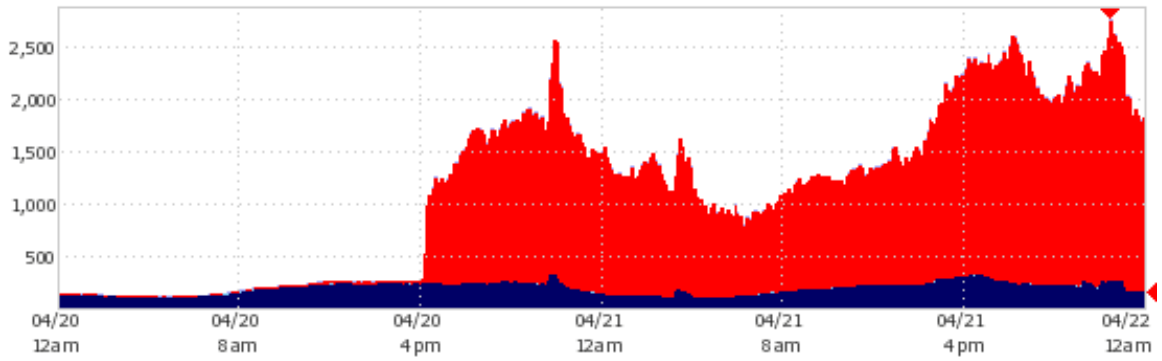
During each iteration, which varies between 2 and 3 weeks depending on team, we allocate 2 pair days to to the Borat Squad. This gives us 3 pair days across all 3 teams to work on shared issues, and allows us to continue practising XP by sharing knowledge across the teams.

Due to the set-up and efforts of the Borat squad we've since solidified our infrastructure across all teams. Everyone runs on the same operating system, uses the same tooling, and deploys in the same way. Reducing variance in this way has allowed us to effectively focus our efforts on increasing overall quality in incremental steps.

Making things visible

One of the most memorable emails we received at Unruly was from our CDN provider. It politely enquired as to whether we were aware that over 80% of our traffic was resulting in server errors. It was accompanied by a graph like the following

[77] https://twitter.com/DEVOPS_BORAT/status/302542701247741954



The plot thickens

There was a moment of panic as we looked to see whether we were still serving ads, and whether we were making any money. It turned out we were serving ads as normal,which is why we had not been alerted by our own monitoring. The problem turned out to be a change we had deployed in our ad-units had caused one particular version of Internet Explorer to miss the end condition of a loop counter and continually request non-existent files.

At times like this there is the temptation to react by adding more checks and reviews to our deployment process to help us avoid releasing things that are broken. Could we have avoided the problem by having more people review a changeset before each release? Possibly. Could we have avoided the problem by having more automated or manual testing? Possibly.

On reflection we realised that this was the wrong reaction. Any given bug might have been caught by these extra steps, or it might not. We tend to only catch problems we've anticipated in our tests. We realised we had not spotted this particular problem because the things we had cared about enough to monitor and test were still working as expected. Our real issue was we had no way of spotting problems that we hadn't thought to alert ourselves on.

In response, we started building dashboards with live graphs showing various aspects of our production systems, not just operational metrics but business metrics such as number of ad impressions, click through rates, even the rate of money flowing through the system. We made these visible by putting them on displays next to our team board. The result of this was we started noticing things that alerting alone would not have told us. For instance, we saw unusual patterns in our traffic when publishers had misconfigured our ad tags, and when we once again deployed a change that broke a subset of browsers we were able to rapidly respond and fix the problem. Our traffic has natural peaks and troughs and as a result it could have been a few minutes before our alerting system would have worked out that there was a serious problem.

The visible dashboards also help us to get synchronous feedback from our deploys. If we are deploying something it's natural to glance at the dashboards just afterwards to see if everything looks OK.

Services/Versioning

Like many organisations, we started out with a couple of monolithic applications. As these grew ever bigger, our deploy speed reduced. Test suites took longer to run, and it became harder to reason about the consequences of a change.

Partly as a reaction to this we started moving towards smaller, interoperating components. At the same time, we were transitioning our source code from Subversion to Git and initially stored each small component in its own Git repo. It didn't take long for this to cause us serious problems with our rapid

release culture. We found we would deploy one component and inadvertently break other components that relied on it. Individual components were now faster to deploy, but we had less confidence in the system as a whole.

We experimented with strict versioning schemes for services, but this led to increased deployment complexity and operational complexity, with a need to calculate the order of changes to be deployed without a single revision identifier of an atomic changeset, and multiple versions of the same service running at a time. We also felt we weren't truly integrating our changes continuously because old code was running in production.

In response, we are now coalescing our Git repositories into bigger repositories which each contain multiple projects. This means we have atomic history again and it also makes it easier for our deploy tooling to work out what has changed and needs to be re-deployed to production. We are also focussing a lot more on monitoring our production system. We choose to omit some acceptance tests on production deploy if it is highly unlikely the deployment will have an impact on a particular system, but there's no reason our acceptance tests can't check our production system.

Delivering major design changes incrementally

One of the most difficult changes to make incrementally is a large architecture transition, which is a problem we have faced and conquered. We serve our ad-units from static JavaScript assets hosted on Amazon S3, which originally pulled down a specific JSON file when loaded to configure video, layout, and sharing information.

This is reliable and allows us to still serve ad traffic when other parts of our infrastructure are unavailable, but gives us almost no control over what happens between the assets and the ad-unit. When our feature set expanded the number of required assets ballooned and required bigger/more files to be uploaded until response times became unacceptable.

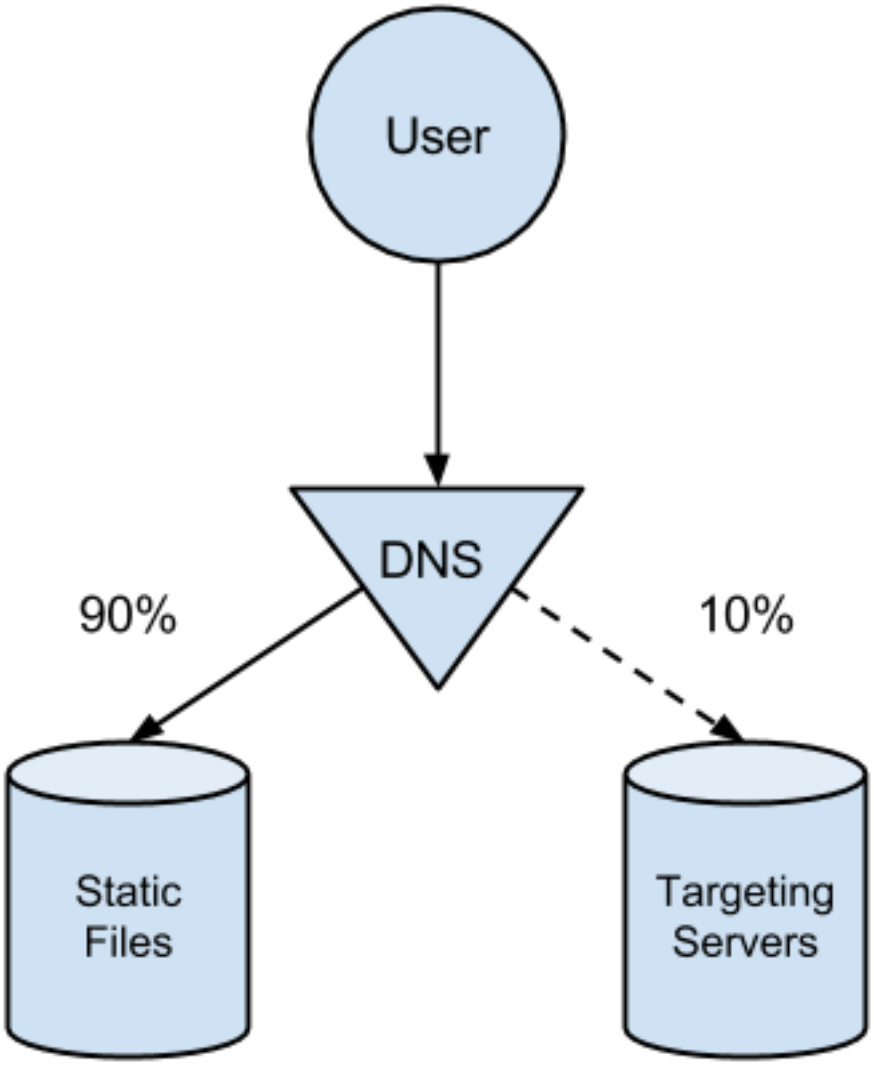
Furthermore, we observed the landscape of the online advertising ecosystem changing, and so we decided to move our entire ad-serving stack to an architecture more consistent with OpenRTB[78]. Since we are consistently serving multiple thousands of ad impressions per second, it wasn't possible for us to simply switch off and over. As we deploy straight to production to get feedback as soon as possible, this meant that we'd have to get the performance characteristics exactly right first time - a risky proposition at best.

In the end, we decided to not change what was being served, just the way we served it - an implementation of Canary Releasing. We designed a set of targeting servers which would at first just transparently proxy requests onto the S3 bucket, and leveraged weighted DNS to help with the migration - since S3 is globally available we needed to do a little work to achieve similar resiliency. We ended up executing this in a 3-stage plan.

1. Set up a new DNS record to point to our static assets
2. Spin up targeting servers in the required regions and add them to the new DNS record
3. Send 10% of the traffic to the new servers and monitor

When we were happy with the performance in production of how we were serving the new assets we would ramp up the traffic by another 10% or so, and leave it running for a few days to expose any performance bugs that we hadn't caught at lower levels of traffic (ulimits - d'oh!).

[78] http://www.iab.net/rtbproject



Branching requests using weighted DNS

Developers at Unruly tend to be fans of testing in production, and deploying new infrastructure changes is no exception - such events are often accompanied by the mantra of "Let's shake the production tree and see what falls out".

After around 6 weeks we were serving all traffic through these new servers and iterating on new features quickly, which we weren't able to do when we were serving static files from S3. Our success with this approach meant that when we recently needed to split our targeting servers into a client/server architecture, we repeated this workflow and were able to turn the change around in just over 3 weeks of work - and most of that time was spent on the actual application development.

Conclusion

Operating as a full-stack team affords us many opportunities to practice both XP and Continuous Delivery at all levels of our software development process - to us there is no distinction between parts of the process that would be siloed away from each other in other companies.

We've had our share of problems during our growth period, but operating in a fashion that emphasises Continuous Delivery means that each time we've been able to conquer our issues and improve, as well

as consistently deliver value to the business at each step along the way.

About the contributors

Alex Wilson *has been a software developer at Unruly for approaching two years, during which time he has had the opportunity to experience and solve scaling issues with their software development methodologies. He takes particular delight in the application of both Continuous Delivery and Xtreme Programming practices to the 'DevOps' process, made easier by Unruly's stance of emphasising the value of generalists over specialists.*

Alex Wilson

Benji Weber *is a developer at Unruly - a marketing technology company - where he has been delivering working software to production almost every working day for the last few years. He has had the privilege of working with a variety of different technologies and teams, experiencing some of the different challenges and opportunities for continuous delivery in different team sizes and tech stacks. Benji has a particular interest in Extreme Programming practices, Continuous Delivery, and Domain Driven Design, along with a wide range of technology interests.*

Benji Weber

We need two DevOps resources, please - Anna Shipman

Anna on Twitter: @annashipman[79] - Anna's blog: www.annashipman.co.uk[80]

GOV.UK[81]

Timeline: *2010 to now (September 2014)*

Introduction

A few weeks ago, a product team at the UK's Government Digital Service (GDS) included in its weekly report the phrase "awaiting decision around getting a DevOps resource". This isn't what DevOps means. DevOps means a culture where developers and web operations engineers communicate and ideally work together, rather than a siloed organisation where developers throw code over the wall to operations and expect web operations engineers to make sure it runs in production. One "resource" can't do all that.

But the fact that a team within government is even looking for someone to do DevOps is a huge step. Government is not the environment you'd imagine as being receptive to a DevOps culture. Traditionally, government favours large, enterprise software projects and is very risk averse. So why is that team even trying to do DevOps?

How we built a DevOps culture at GDS

It's first worth explaining how GDS was started in the first place. In 2010, Martha Lane Fox (at that time, the UK's Digital Champion[82] wrote a report[83] presenting a new vision for the future of digital in UK government. Her recommendations included creating a new, central team to have control over the user experience across all digital channels, and making a "government front end for all departments' transactional online services".

In 2011, a team of twelve people put together an Alpha version of what a single domain for government would look like. A few weeks later, the team was called back and asked to create a Beta of the same thing. The team grew in size, but there was a lot to do and it was recognised early on that infrastructure, operations, and compliance were non-negotiable. While you can reduce scope of features if necessary, there is a level of infrastructure that absolutely has to be in place. So, as with many of the other work done at this time, someone just picked it up and started doing it.

Initially, the Beta was private. Around January 2012, the decision was made to go into public Beta. At this stage the team was around 120 people, split into several product teams, and the deployment process I describe below was already in place. As we prepared to remove the authentication for the public Beta, one of the directors nervously asked whether the infrastructure team were sure that deployments would work. "I think so," came the response. "We've already done over 1,000."

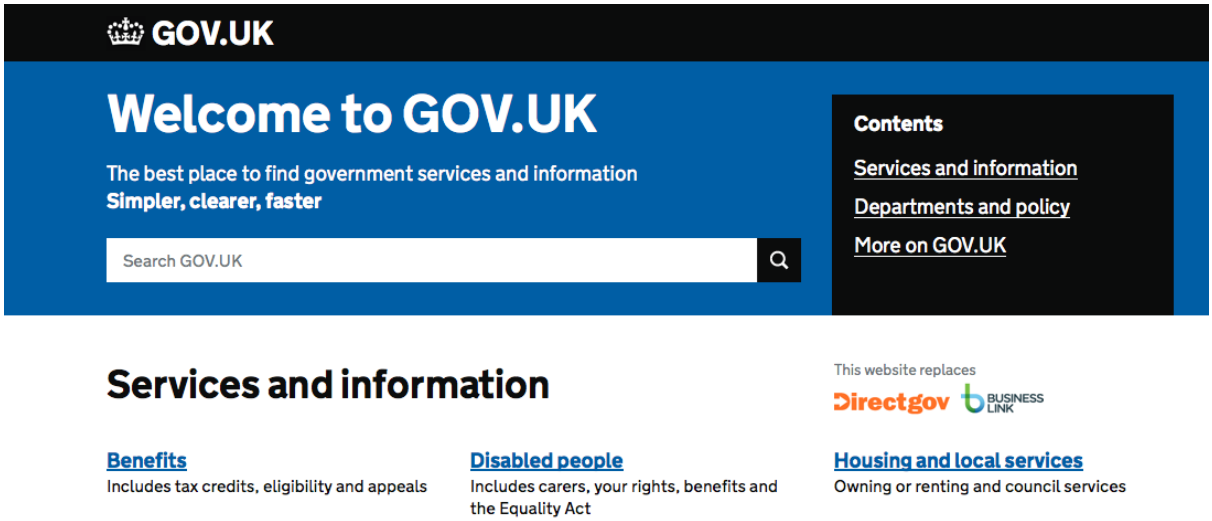
[79] https://twitter.com/annashipman
[80] http://www.annashipman.co.uk
[81] http://www.gov.uk/
[82] http://en.wikipedia.org/wiki/Digital_Champions
[83] https://www.gov.uk/government/publications/directgov-2010-and-beyond-revolution-not-evolution-a-report-by-martha-lane-fox

Essentially, the way we developed a DevOps culture was the same way we developed many other parts of GDS – by being trusted to try it and then by proving it worked. For example, developers using their own laptops to deploy to production was a big win for us: initial plans to ensure security involved a special, locked-down laptop, only a few people permitted to do the actual deploys, and deployments far less frequently. However, because we spent so long in Beta, we were able to prove that developers had the skills and responsibility to manage continuous delivery.

We were also able to demonstrate that we had an understanding of the risks of this approach and took security of the site seriously. We arranged workshops with GCHQ[84] (Government Communications Headquarters – the British intelligence and security agency) and others to review the architecture and identify particular risks and attack vectors, and made adjustments based on them. This helped a lot with building relationships. We tried to keep the focus on the security-related arguments for our approach (simplicity of rolling forwards, ability to patch things out of hours, credential management) rather than making it about our own convenience or preferences.

GDS and the GOV.UK website

GDS is now around 600 people, working in a number of areas, of which the GOV.UK website is one. For example we also work on transforming government transactions to make them digital by default and changing the way government buys IT services to waste less money.



The GOV.UK website

GOV.UK is a service-oriented architecture, made up of small cross-functional teams that work on individual applications or areas. For example, the *Search and Browse* team works on improving search and navigation on GOV.UK, especially important given the large amounts of specialist content that is coming onto the site as more agencies transition on. Each team is multi-disciplinary, so a standard team might have two frontend developers, two backend developers, a user researcher, a product owner and a delivery manager. Some teams have business analysts, some have a dedicated person to work on quality assurance.

Some teams at GDS have their own web operations engineers, but for GOV.UK we have a central *Infrastructure and Tools* team, which I work on. There are currently nine of us – mostly web operations engineers, with one developer and one business analyst. The site runs on about 80 servers, and at the time

[84] http://www.gchq.gov.uk/Pages/homepage.aspx

of writing, gets around 5-6 million hits a day, though this will continue to increase as we complete the transition of government agency sites onto GOV.UK.

Our DevOps culture

I will now explain some of the DevOps practices and processes we have at GDS.

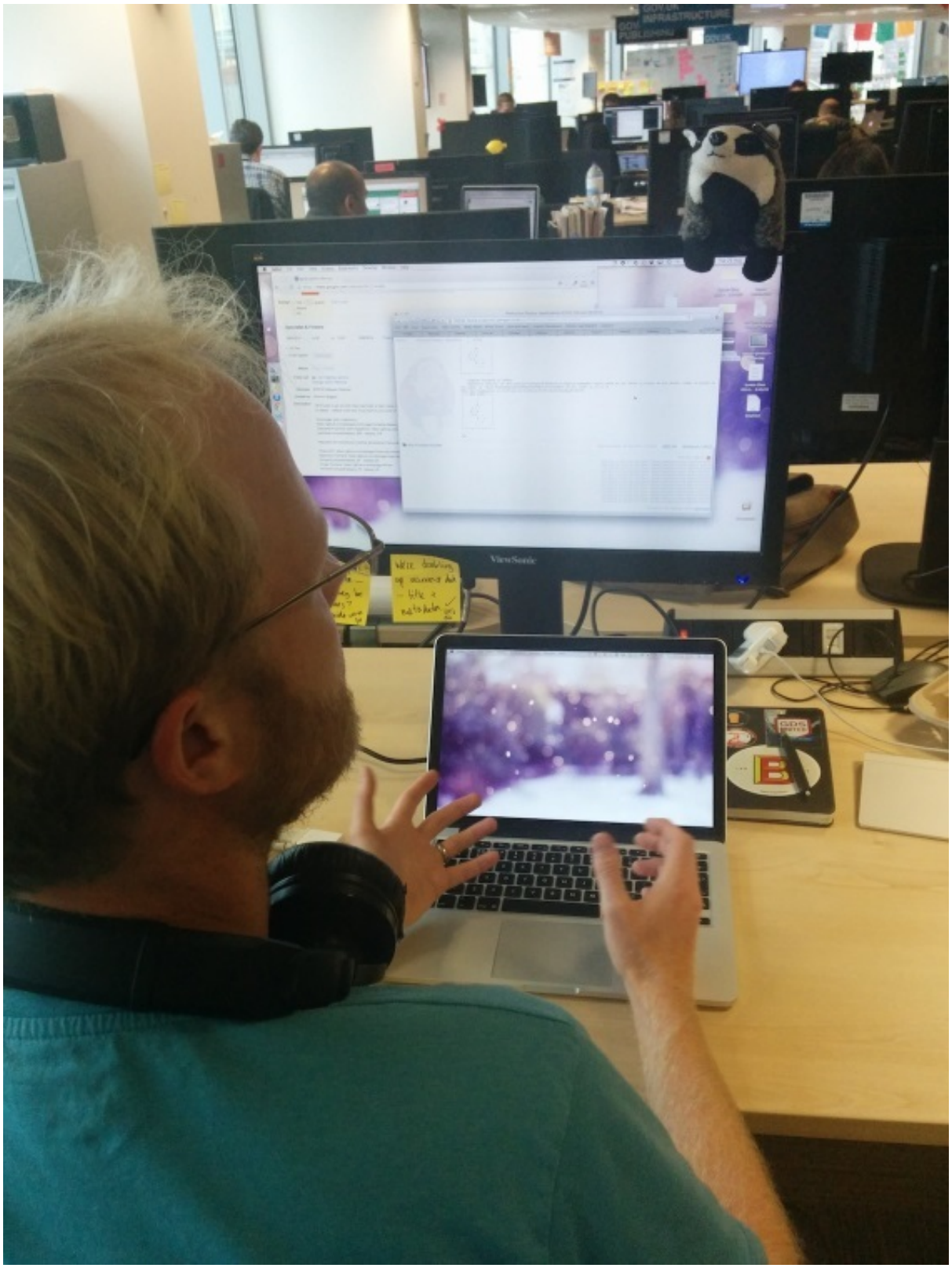
Developers deploy their own code

We release little and often. When a team want to deploy a feature (or features), they book a slot in the release calendar, then come over, take possession of the badger[85] (our change manager is a stuffed badger, a token to ensure that only one deploy happens at a time), and then kick off the Jenkins job that (mostly) uses Capistrano to deploy from the GitHub repository of whichever application they are deploying. We deploy several times a day, which is quite a change for a culture used to six- to eighteen-month release cycles.

The Badger of Deploy © Dafydd Vaughan

[85] https://twitter.com/badgerofdeploy

It is up to the team deploying to test their code on the staging environment, then babysit it into production and make sure the GOV.UK smoke tests pass once the feature has been deployed. In addition, they are expected to be available to address any issues that should come up, so they need to be sensitive to the scale of the change they are making. The release calendar is from 10am to 4pm on weekdays, so there are always people in the office to deal with issues and revert if necessary.



The badger watching over a deploy © Daniel Roseman

Our 2nd line model

As well as deploying their own code, developers on GOV.UK also support their code in production.

The current '2nd line' model is: on a rota, one developer at a time sits with the Infrastructure and Tools team for a week. One developer and one web operations engineer are on 2nd line support for that week. The role of the developer is mainly to be an 'interrupt monkey' as described in the excellent book Time Management for System Administrators[86]. They keep an eye on our dashboard of alerts (we use Icinga for alerts and have a dashboard showing the current state of health of our production, staging and preview environments) and our Graphite dashboard showing various graphs. If an alert fires PagerDuty, their phone is the one that rings.

They also help with deploys; for example, contractors on GOV.UK are unlikely to be security cleared, and you need security clearance to be given production access, so if a contractor needs to deploy code they will sit with the 2nd line developer and they will pair on the deploy. The developer is the primary on the 2nd line rotation, and the web operations engineer is there as first point of back-up call if the developer needs assistance.

Because monitoring alerts and doing deploys doesn't take up a full working day, the developer usually works off their own team's backlog in between times. It is a good time for developers to tackle any operations stories in their backlogs, as sitting with the infrastructure team means there will be plenty of us around who can help.

We now have a lot of developers on 2nd line and are in the process of moving to a new model where two developers will rotate on at a time and this team of three will work off a 2nd line backlog. Inspired by Etsy's bug rotation[87], this backlog will be made up of bugs and support issues. This will give us a chance to address bugs that are not a priority for teams working on products but still will deliver value to the user, and will also give developers a chance to work on projects they've not worked on before, so sharing the knowledge around.

Our on-call model

Most of the developers and all of the web operations engineers who populate the 2nd line rota also take turns on the on-call rota. The on-call rota applies from 5.30pm to 9.30am (overnight) and all weekend, and two people are on the rota at a time. The primary will get called first during weeknights and second at the weekend, and the secondary vice versa.

[86] http://shop.oreilly.com/product/9780596007836.do
[87] https://www.youtube.com/watch?v=pr3rBFC7Tp8

3	Week	Date Beginning	Date Ending
4	97	2-Jul-2014	8-Jul-2014
5	Phone Number	Current Assignment	Duty Role
6	[redacted]	Bob	Primary In Office
7		Rob Y	Secondary In Office
8		Jenny	Primary Weeknights Oncall
9			Secondary Weekend Oncall
10		Tom	Secondary Weeknights Oncall
11			Primary Weekend Oncall
12		Albert	Ops Manager
13			

A week from the rota showing roles

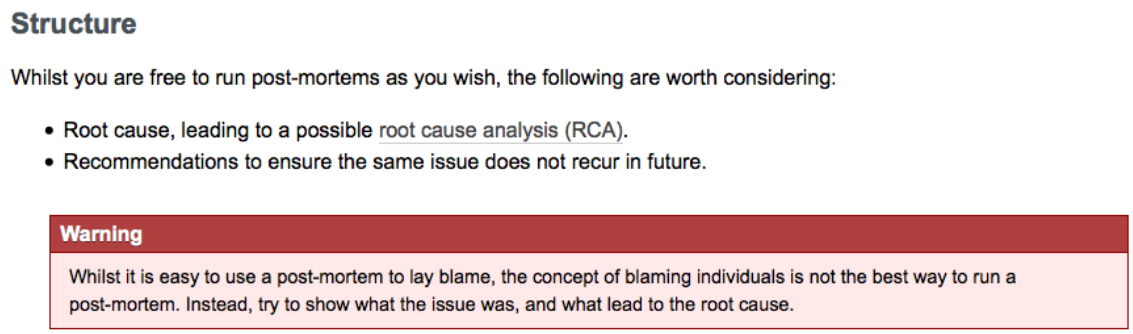
Only crucial alerts fire PagerDuty, which will call the primary phone, and if the alert is not acknowledged after a certain time, the secondary phone. There are also requirements on us to put information on GOV.UK in certain types of national emergency, and if that happened out of hours then the on call phone would be used for that.

If you are the one who is going to be woken at 3am to deal with an alert, there's much more of a motivation to make sure the alerting is good and relevant, and we are continuously refining the alerting so you only get woken up for something you can fix. When you know it's you who will be on the hook, there is more motivation to spend your 2nd line rotation learning about what the alerts mean, how to address them and perhaps fixing the underlying issues.

Incidents and post-mortems

We always arrange a post-mortem when there has been a production incident of some kind. We identify the root cause and make recommendations to prevent another occurrence. We also write and circulate an incident report, which includes at least a timeline, estimated number of users affected and the recommendations for next steps.

Invitations to the post-mortem are circulated widely: as well as stakeholders of the team or service in question and those involved in the incident's resolution, the invitations are sent to a number of email lists. Anyone is welcome to attend, leading to a more varied and effective set of recommendations, and ensuring that we are being honest and transparent about the process. Incident reports are also circulated widely and are available to be viewed by anyone at GDS.



Structure

Whilst you are free to run post-mortems as you wish, the following are worth considering:

- Root cause, leading to a possible root cause analysis (RCA).
- Recommendations to ensure the same issue does not recur in future.

Warning

Whilst it is easy to use a post-mortem to lay blame, the concept of blaming individuals is not the best way to run a post-mortem. Instead, try to show what the issue was, and what lead to the root cause.

A screenshot from the opsmanual page on running post-mortems

We do not look to apportion blame in post-mortems; they are solely for finding out what the root cause was and what we can do to prevent it. This is made very clear in the opsmanual.

The opsmanual

We have a very detailed operations manual ('the opsmanual') which we keep up to date. This has detailed instructions on how to resolve particular alerts and also includes the step-by-step process for emergency publishing. It is very much living documentation and developers often update it when on 2nd line. When people ask questions about how to do something when on 2nd line, the answer is either already on the opsmanual, or they update it when they've found out.

This hasn't been without its challenges. We do not have a single source of information at GDS, as different teams prefer different wiki software. For example, most of the HR information is on a Google Site, called the 'HR wiki', but this does not provide the searchability or commit history we would like for the opsmanual. The opsmanual is a Sphinx project (the pages are written in reStructuredText), hosted on our internal GitHub Enterprise installation, which we use for private projects, like our Puppet repo. (Most of GDS's code is hosted[88] publically[89] on GitHub.) However, you need a licence to access GitHub Enterprise, and these are costly and limited, so non-technical staff often do not have access to even view the opsmanual. While most of the content is extremely technical, it would be useful, for example, to be able to publish our incident reporting guidelines and post-mortem process more widely.

Product team outreach

One other way we support DevOps at GDS is occasionally, a member of the infrastructure team will go and sit with a product team for a set period of time. Sometimes this is for a few weeks to help them with a number of issues or to get started with web operations, and sometimes this is just to help with one issue. For example, recently two of us helped the Transition Tools team migrate from MySQL to PostgreSQL. This means teams rarely introduce a major new piece of software without a proper cross-disciplinary conversation about how to manage it.

And Tools

I mentioned before that we are the Infrastructure and Tools team, and so far I've mostly been talking about infrastructure. We also build tools. When a recruitment consultant is looking for a "DevOps Engineer", they tend to be looking for a web operations engineer who writes Puppet/Chef/Ansible rather than hand-crafting snowflake servers. Well, we also do that kind of "DevOps".

[88] https://github.com/alphagov/
[89] https://github.com/gds-operations/

We use Puppet to manage configuration of our machines, and we build tools to automate whatever we can. For example, I have been leading a small team for a few months building vCloud Tools[90], Ruby gems that automate provisioning of our machines in VMware's vCloud Director, which is the cloud infrastructure of the platform our production environments are currently hosted on. Where we can, we automate, and share our automation with other teams in GDS and in government.

The challenges we've faced

I've mentioned a little about the challenges around the sharing of information, but we have faced some other difficulties putting our processes into place.

The 2nd Line model was initially a hard sell

As we neared our launch date of October 17th 2012, all teams were working flat out to meet the minimum required for us to be able to switch off the existing Directgov and Business Link sites (which were to be replaced by GOV.UK), and product owners pushed back on the idea of giving people up to work on 2nd line. In addition, the team working on moving departments onto GOV.UK had a later release date and did not feel in a position to think about operating, let alone spare developers for a week at a time.

There were also problems for developers. Not everyone could be on 2nd line, because we had grown so fast that many people were new and we were very dependent on contractors, but production access requires commitment to GDS, a track record here and security clearance. This meant that shortly after launch there were a few teams where there was only one developer who could be on the 2nd line rotation. In one case the developer in question was also the tech lead, so her team was disproportionately affected when she was sitting with us.

Developers and product owners eventually warmed to the 2nd line model as they saw the benefits of understanding how the code works in production and how to improve things by taking shared ownership, as well as how it reduced interruptions to the product teams. Our reduced reliance on contractors and recruitment of more permanent, committed employees also helped. But it was by no means a smooth process.

The 2nd line model requires a lot of operational support

The operations manager does a lot of work to support 2nd line. He puts a lot of effort into the rota to minimise disruption to people and teams, for example making sure two devs from same team are not off at same time, making sure people are not on 2nd line at the same time as they are on-call, evenly spacing people's rotations and pairing people new to the rotation with those who have more experience. He also manually forwards the support phones to people's work phone numbers each week, rather than giving them another phone they have to carry around.

The current model of one developer and one person from the infrastructure team means that it requires a lot of support from my team as well, as we are on 2nd line more frequently (though as the secondary, so we are not the first port of call). The 2nd line person also sits with our team, which means we get involved in more things, and also that we de facto own it.

The new 2nd line model, where two developers will be on per rotation and will work off a backlog of bugs, supported by a web operations engineer, will mean the load is more fairly distributed among teams. In addition, the 2nd developer will not need to have production access so contractors, juniors and newer members of staff will be able to take part.

[90] https://github.com/gds-operations/vcloud-tools

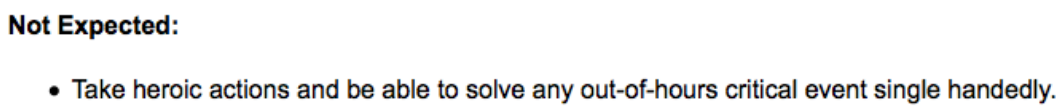
The on call model is mostly voluntary

For people who joined before late 2013, the expectation that we would join the on-call rota wasn't in our contracts which means that for many people it is on a voluntary basis. There is a small extra payment for taking part, but it is extremely small; something in the region of £6 a day (around 10 USD) and time off in lieu if called to deal with an incident. In return, you need to be able to answer your work phone, be sober, and be within 20 minutes of your GDS laptop.

This means it is dependent on trust and goodwill, as well as confidence working in production. For a while the rota was very heavily populated by the infrastructure and tools team, with three people in particular taking a huge share. In the first 26 weeks after launch there was only one week when none of those three were on call. Now it is much more evenly distributed, and while there are a handful of developers still not on the on-call rota, most are. You can expect to be on call once every couple of months.

An advantage of the mostly voluntary nature is that we have to be very careful about what is permitted to call the out-of-hours phone, because if on-call became unreasonably onerous, many people would opt-out. The operations manager has fought very hard to not pass the phone number out to departments, who are very keen to have a number they can call for perceived emergencies. Only PagerDuty calls the on call phone. All other out-of-hours calls, including national emergencies, go to the operations manager, and he makes the required assessment of whether to call the on-call person.

Many developers are new to being on call, so we have evolved a very clear process, particularly to help those nervous or inexperienced. The opsmanual details exactly what you might be expected to do with step-by-step instructions of how to do it, and more importantly, what you are not expected to do. The list of what you are not expected to do starts with "Take heroic actions and be able to solve any out-of-hours critical event single handedly."


Not Expected:

- Take heroic actions and be able to solve any out-of-hours critical event single handedly.

A screenshot from the opsmanual page about what is expected when on call

We've also done a lot of work to make sure the site is robust. It is extremely rare that anyone is woken by PagerDuty; if it were more common, it is unlikely that this model would survive. We also have an advantage that we do not provide 24/7 application support. In most cases, the out-of-hours action would be to fail the site over to the static mirror. Because the site is so cacheable, the majority of citizens would not experience any difference.

Recruiting into the civil service is hard

One hurdle we have to jump quite frequently is civil service recruitment. Applying for a civil service job is very different to applying for a job in the private sector. It is very important that the competition for jobs to be paid for by tax money is open and fair, but in order to ensure this, the process is very arduous.

When I applied, I had to fill in a 9-page application form. Now the process is slightly simpler – you send in your CV and a statement of suitability for the post; but the statement of suitability is absolutely essential and must give evidence of how you meet the Civil Service Competencies[91]. To help people understand how to apply, I wrote a blog post explaining how to write the statement of suitability[92], and we publicise it as widely as possible when we are recruiting, but candidates that otherwise look suitable can sometimes fail to progress because of they have not understood this requirement.

[91] http://resources.civilservice.gov.uk/wp-content/uploads/2012/07/Civil-Service-Competency-Framework-Feb2013.pdf
[92] https://gdstechnology.blog.gov.uk/2013/12/24/applying-for-a-job-at-gds/

While this does mean we miss out on candidates who look very good, one small compensation is that the application process does select for people who are able to communicate ideas clearly, which is essential for DevOps. It also does make the process more fair than in the private sector. It's not just about who you know or what you look like – for those who progress to interview stage, a panel of three people will ask each candidate the same set of questions and score them separately. This does mean that there is less chance of government money being given away on favouritism.

These sorts of open and fair rules apply to other things as well, for example buying IT services. This chapter is not long enough to go into more detail about that: suffice to say, it is also very time-consuming and rigorous, but this is also to make sure it is fair.

Culture clash

One of the ways that GDS has grown to 600 people is by absorbing teams doing similar work elsewhere in government. Those teams consist mainly of people who have been civil servants for a long time, and often come from a more corporate civil service background which sometimes causes a clash of cultures within GDS. One source of this is around language, for example the use of language that is on our list of banned words[93]. And this explains the kind of misunderstanding I opened with. We handle this in GDS by having a style guide we can point to, by extensive use of the Government Service Design Manual and also just by having those conversations with our colleagues.

While this approach seems to work within GDS, this highlights one of the largest challenges we face. I've left this one to the end, because we've not yet addressed it as well as we could have done by now. On the GOV.UK team, we have not integrated terribly well with the rest of the civil service. Compared to many London-based civil servants we go to very few meetings in Whitehall, and when we do we very clearly stand out: we wear jeans, our laptops are Macs, and are covered in stickers. There is definitely a sense in which other departments sometimes think we are arrogant, and this doesn't help us get our message out.

[93] https://www.gov.uk/design-principles/style-guide

“My first GDS Mac” © Ben Terrett

Outside of the GOV.UK team, GDS has addressed this problem better. For example, we have built up good working relationships with the procurement teams and with security teams from GCHQ. Some of our colleagues have spent long periods of time embedded in departmental teams, for example in DVLA[94] (Driver and Vehicle Licensing Agency). However, on the GOV.UK team, we need to do better.

This is not a DevOps issue as such, but it is a problem for DevOps because that is one of the important ways of working that we want to share, and it would be a shame if the GOV.UK team missed that opportunity. We are aware that this is a problem and recently have started addressing it, for example by making an effort to send people to civil service events. Realistically, the civil service is extremely large (around 400,000 people), and spreading the approach is difficult, but we are keen to do so, and I hope we will continue to improve the way we handle this.

What can someone reading this take back to their organisation?

The main take-home messages I’d like to leave you with are

1) learn from us:

- the best way to bring in a new process is to show it working, and demonstrate that you understand the risks implicit in that process. Trust and goodwill go a long way.

[94] https://www.gov.uk/government/organisations/driver-and-vehicle-licensing-agency

- communication is important. Documentation is very important.
- it's great for developers to support their code in production, and having them sit with the infrastructure team during working hours regularly is a good way to ease into it.

2) do better than us:

- when you are making big changes to business processes, sometimes you have to go out of your own comfort zone to bring the business along with you.

Coda

Back to the team looking for a "DevOps resource". In the intervening week, at least one person had a chat with them. Their next weekly report said something different: they were "awaiting a decision around getting a person who will be focused on the culture that is DevOps[95]". We're getting there.

About the contributor

Anna Shipman *is a senior developer at the Government Digital Service. She works on the infrastructure team that supports the GOV.UK website, doing development and web operations. Currently she is working on open source tooling to provision VMware vCloud Director environments: https://github.com/alphagov/vcloud-tools. She works mainly in Ruby and Puppet at the moment, though her past includes Python, Perl, Java and the JEE stack. She blogs at www.annashipman.co.uk, tweets at @annashipman and is always up for a game of pool.*

[95] https://www.gov.uk/service-manual/operations/devops.html

Process kick - Amy Phillips

Amy on Twitter @ItJustBroke[96] - Amy's blog testingthemind.wordpress.com[97]

Songkick[98] - Songkick developer blog devblog.songkick.com[99]

Timeline: February 2011 - August 2014

Songkick was founded in 2007 to help music fans find out when their favourite artists are coming to town and get tickets. With over 10 million unique users per month Songkick is currently the second most used live music service in the world after TicketMaster.

Currently employing 30 staff in London and Portland, Oregon. Songkick exists on the web for fans (Songkick.com) and for artists (Tourbox.songkick.com), on mobile (iOS, Android), and as a Facebook app. The Songkick API is used to integrate concert listings into numerous third party applications including Spotify, YouTube, and FourSquare

The decision to change your release process should never be taken lightly. Releases are risky and require co-ordination throughout the team and any successful process change requires a change in culture. But attempting to change either process or team culture is a huge undertaking, and creating a pipeline to automatically release code to Production sounds like an very challenging task.

Back in 2011 Songkick made the decision to take on this challenge.

Needing a change

At the time we were a small development team with big ideas. Our team consisted of 7 developers, 1 frontend designer, 1 User Experience (UX) designer, 1 Product Manager (PM) plus myself as the sole tester. We were working to create and maintain the Songkick.com website which provides live music alerts to fans worldwide, and the Tourbox website which allows artists to manage their tour dates and display them on 3rd party sites such as YouTube and Spotify. We also had an iOS app, an Android app, and a Facebook app.

At Songkick we have always believed that new features should be shipped to the user as soon as they are ready. A new feature only starts providing benefit when it is being used by real users, and we're a startup; trying to find market fit involves quickly validating ideas. We want to understand who our users are and what problems they want solving. No matter how closely the Songkick staff represent real users - and they are definitely all real users - nothing compares to seeing a new feature being used across the world by millions of fans and artists.

At the time our process involved the PM defining a new feature, such as a new page footer to encourage signups or a feature allowing users to import their listening history from Spotify. The Design and UX team would prototype some options and create visual designs. A developer would implement the feature and then we would do some testing. Once the feature was live we could monitor usage and identify feature improvements.

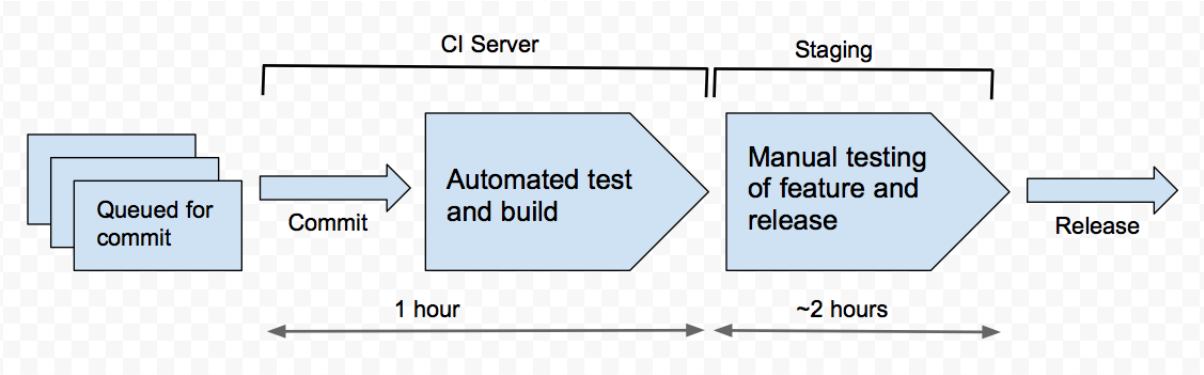
[96] https://www.twitter.com/ItJustBroke
[97] http://testingthemind.wordpress.com
[98] http://www.songkick.com
[99] http://devblog.songkick.com/

This approach succeeded for some time. The team works well together and by involving all the different roles we were able to design, implement, test and ship new features within several days. However, the problems started as Songkick scaled up. With more developers we were able to build more features and slowly we realised that our release process was a serious bottleneck.

Each release required both regression testing (particularly around frontend code) and feature testing to ensure PM and Design team expectations were met. We ran test suites of unit and integration tests over multiple EC2 instances to keep our total build time under an hour, the downside being flaky builds which often required re-runs. Once on our Staging environment it was accepted that we would need at least one bug fix - and that bug fix would necessitate a re-build and more testing to look for new problems caused by the fix.

Like many development teams, we did not explicitly design our build and release process. We had a Git repository and Jenkins managing Continuous Integration but mostly we responded to problems:

- When a release accidentally went out without the final copy we decided all releases needed a copy sign-off.
- When a release didn't meet design expectations we decided all releases needed a design sign-off.
- Problematic releases which contained multiple features caused extra stress - which change had caused the problem? Were other features in the release delayed because of this one thing? Often the cause of delay would be a trivial bug fix implemented as a personal project, particularly frustrating and difficult to predict for the non-technical team. Eventually, we decided that each release should only contain code from a single story.



The Songkick release process including average build and test durations

On and on it went until we reached the point where each release was pretty predictable but we were severely limiting our freedom to make a release. Queuing up releases causes obvious delays in getting code out to users but it also impacts the development team's ability to improve code. At the same time as our releases dropped off, with even trivial features taking 1-2 weeks to release due to queueing, we were hiring to expand our development team. It was incredibly frustrating to have to admit that we had reached our release limit - adding new developers was not going to help us move faster, because we couldn't get completed features out to users.

Something had to change.

Up to that point we had released features as soon as we could. We knew that we needed to get features out to users as soon as possible but we delayed ourselves by aiming for perfection. Infrequent releasing encourages this behaviour by making it difficult to predict when the next release, and potential bug fix, will go out.

We needed to decide between scheduled releases which would give us control and predictability or a continuous release approach with less control. Scheduled releases sound simpler; everyone knows exactly when the release is expected and there is a clear shift in focus as you move from feature development into code-freeze and testing. However it was clear that regression testing was already a bottleneck. We knew that scheduled releases would tie up all testers until the release date, at the same time the developers would likely be starting to code new features ready for the next release. It felt like we would be creating an unhealthy divide between the developers and testers.

Trusting that 'bringing the pain forwards' would teach us how to release code painlessly, and hoping that being able to release when we needed to would help us overcome our fixation with perfection, we decided that a careful adoption of Continuous Deployment could overcome most of our difficulties. Our goal was the unblock our release pipeline in the simplest possible way.

Making the change

After several months of researching release approaches, and talking to other companies who had made similar shifts to Continuous Deployment we were able to begin our transformation. Most of the team were on board to try the change, we had reached a stage where releases were so difficult and so infrequent that it was clear we needed to do something. Almost everyone was fully committed to taking the leap to Continuous Deployment rather than just CI. We knew this would require additional monitoring and make risk harder to manage, so we decided to focus on small steps towards Continuous Deployment.

First up was removing the integration queue and for that we had to solve our unreliable builds. When the development team was small only one developer was integrating code at any one time, and flaky builds were easy to spot. However, as the team grew, broken builds became a serious problem, with developers checking in code on top of broken builds and unrelated incomplete features causing regression issues in one another. To encourage a cultural shift we used annoying (but effective) sound effects to make it known to the whole office when a build was broken. From there we were able to analyse breakages and make changes to improve stability.

Seeing Progress

After a few weeks we were in a great position - we had moved the bottleneck, a strong indicator that you have succeeded in making changes. Instead of having code queued up waiting to be committed we now had too many builds ready for testing. Each release into our Staging environment still required design and copy sign-off as well as feature testing and regression testing. New features were still suffering from inevitable bugs and tweaks which blocked releases and wasted time by duplicating regression testing. Testing took so much longer than our build process that we had multiple green builds sitting on the shelf waiting for testing.

We decided to try and shift as much testing as possible to the front of the pipeline. By bringing the developer environments inline with the Staging environment we were able to perform the majority of feature testing upfront. Moving design and copy sign-off prior to code commit meant that stakeholders did not need to be present for a release.

Focusing on building small pieces of functionality, either as iterative development, or as simplified features, taught us the strength of treating feature launches as separate from code releases. Feature Flippers[100] gave us the tools we needed to make committed code safe to go into Production. If delays

[100] http://martinfowler.com/bliki/FeatureToggle.html

were to happen, either because of quality issues, or availability problems, we were only delaying a single developer and a single feature rather than blocking the entire company's release pipeline.

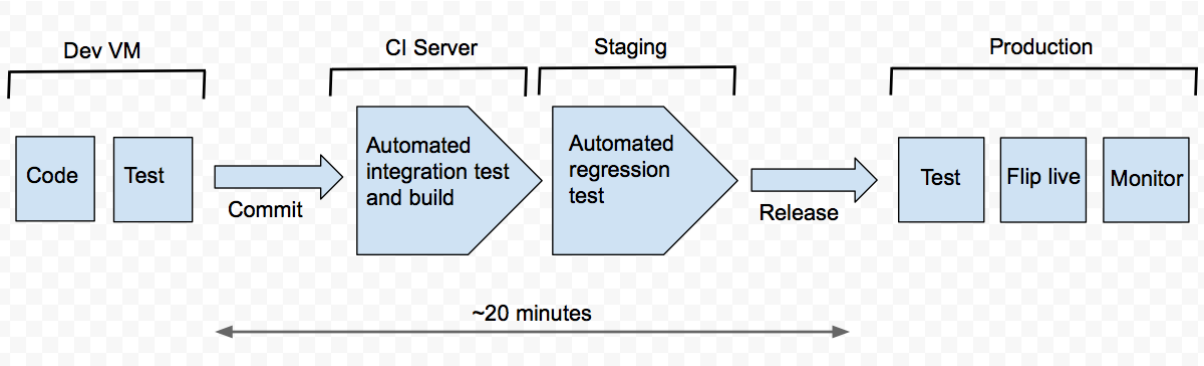
Seeing the benefits of this process change motivated the developers to make improvements to our automated test suites. Previously we were using Cucumber and had an extensive set of high-level tests. Touching multiple layers of the stack, and in particular the database caused the tests to be slow. Once we realised how difficult it was for developers to run these slow automated test suites locally we prioritised the work to speed the tests up.

From talking to other teams who were successfully using Continuous Deployment we knew that our test coverage was above average. Just as with our features, we were trying to cover every eventuality and falling into the trap of trying to test everything. Despite this we still relied on manual regression testing, and still missed bugs which could have detected with automated checks.

Analysing exactly what we were checking, and comparing this against the bugs we found in Staging, and more importantly the bugs which users contacted us about allowed us to understand the gaps in our coverage. By considering the risk of different parts of the website, looking at the impact on the end-user, and using this to focus testing on the high and medium risk areas we were able to simplify and speed up our test suites. Many of the checks were rewritten as lower-level unit tests, which meant a focus upon small chunks of code rather than entire user journeys. Our slower running Cucumber tests became focused on journeys we, and our users, actually cared about. Finally we seemed to have reached a place where Continuous Deployment might be a possibility.

Dealing with uncertainty

The non-technical members of the Songkick team were understandably concerned about Continuous Deployment. We were used to working in a way which allowed for plenty of manual checking. Our designers would craft beautiful frontend designs, we would try to give users the best possible user journey, and each word of our copy was precisely chosen. Now it seemed we were proposing a radical approach of simply writing code and pushing it live. One way we addressed these concerns was with a test suite of business defined automated user journeys, and we agreed we would never release if any of these journeys was broken. To make the checks as realistic as possible we used Selenium and Capybara to execute them, via a web browser, against our Staging environment. This gave us enough confidence in a build to consider releasing to Production without extensive manual testing. Suddenly after months of improvements we had reached Continuous Deployment.



Releasing with Continuous Deployment

One of the challenges of Continuous Deployment is handling the cases where you need more testing than is feasible on a developer environment. Often this is caused by environmental limitations, particularly

for performance or security testing. We tried to avoid stopping our release pipeline by testing high risk changes in Production. Feature flippers mean we can run the code on live servers, using live data, but test them as if we were working on a test environment.

Keeping the Peace

One benefit of having a fast release process is the ease with which issues can be fixed in Production. To help us detect issues as quickly as possible we significantly increased our Production monitoring, including adding alerts which would fire if the number of errors on the Production servers increases significantly.

Each day the development team receives an email summarising all Production errors from the previous day; our goal is to reach an empty email.

Using methods like this to visualise blockers or smells is an effective way to keep each other accountable. Together we share the goal of investigating and fixing issues.

Did it work?

Our original goal was to unblock our release cycle. After several months of dedicated work to improve our code, tests, environments, and culture we reached a point where developers no longer faced commit or release delays. We had advanced to a place where we could release code even if a key person was unavailable at the time of the release, we knew how to recognise and respond to issues, and our process openly acknowledged and considered the level of risk.

At Songkick a quality feature is the right feature. We need to understand who our users are and we need to build the right feature for them. The speed and ease of Continuous Deployment provided the unexpected benefit of allowing us to embrace experimentation. We can quickly build a ‘good enough’ version of a feature, release it, and then within a day or two we have enough data to decide if the feature should be implemented for real. Every experiment must have a defined end date, nothing can live on indefinitely until we see the results we would like to see, and the code experiment must be deleted and implemented as live, maintainable code. This clear distinction helps us define the amount of design and testing these experiments need.

At the beginning of 2011 we were in a difficult place, struggling with an unwieldy process and codebase. By reviewing what we were doing, and really considering why, we were able to make small incremental steps towards a better place. Keeping in mind our end goal, no matter how ambitious it seemed at the time was the key to working through difficulties while guaranteeing that we arrived at the right destination. Most significantly of all we learnt how to identify and implement improvements in our process and our products.

About the contributor

Amy Phillips *is Test Lead at Songkick, a start-up created to help you track your favourite bands so you never miss them live. She has spent the last 10 years testing in a variety of development environments for companies including Royal Mail, The Guardian, and Yahoo! After experiencing the value that a well-managed agile team can bring to software delivery Amy became passionate about adapting testing and quality techniques to work within a lean start-up environment. She strives to enable development teams to deliver quality, user-oriented products within the shortest possible time.*

Amy Phillips

Scrum for Ops teams - James Betteley

James on Twitter: @jamesbetteley[101] - James's blog: devopsnet.com[102]

Fourth[103]

Timeline: October 2012 to present (October 2014)

Fourth are one of those companies that people are always surprised to have never heard of before. They're the world's leading provider of cloud-based solutions for the hospitality industry. This means they're pretty big, and since they've been around for quite a while, their systems are pretty big too. Looking after these systems is a very demanding job, and not for the faint of heart.

I joined the Infrastructure Department back in the autumn of 2012, with grand visions of automating the socks off everything and making the hard stuff look easy. It wasn't long before I realised that there was a very good reason why the teams hadn't done all of this stuff already – they were too busy. Yes, they were too busy to do the stuff that would help them to be less busy. To make matters worse, there was a backlog of other work to get done, as well as the day-to-day fire fighting. But, as is so often the case, the problem wasn't technical in nature, it was about how the team was working – you could say it was more cultural than technical.

All Hands on Deck!

The Infrastructure Department was split into 2 teams: the IT Ops team who looked after all the office based systems including test environments (as well as doing desktop support), and (somewhat confusingly) the Infrastructure Team who supported the live environments (webservers, database servers, load balancers etc). Both teams were always frantically busy. Most of the time they were busy doing what I call fire-fighting. They were responding to issues in the live environment, or issues from within the office itself. They were constantly being approached by individuals who had some urgent matter they needed resolving right there and then. Somewhat unsurprisingly, these issues would get addressed very quickly. In essence, the team were subconsciously prioritising these interruptions over the work already in their backlog.

What the team needed was a way of visualising their workload, prioritising it, and somehow be able to give accurate predictions about when it would get done.

Is it Scrum?

Where possible, I try to adopt a Kanban approach within Infrastructure teams, because it embraces the reality of their work, namely that "interruptions happen" and the future is unpredictable. However, the business needed firm dates for the completion of certain projects on the Infrastructure Department's backlog, so I started to look for an alternative approach – something that embraced regular interruptions but also gave us some predictability.

Scrum offers a fantastic framework for delivering software. Delivery teams are generally ring-fenced (and therefore immune from interruptions) and once they've attained a sustainable velocity, they can

[101] https://twitter.com/jamesbetteley
[102] http://www.devopsnet.com/
[103] http://www.fourth.com/

make accurate predictions on how much work they can get through in any sprint, iteration or even a whole release. I wanted this level of predictability, but I couldn't ring-fence any of the team, so I knew the interruptions would continue to happen and impact everyone. I desperately needed to get a handle on the amount of backlog work the team could complete each week, but it varied so wildly from one week to the next! So we decided to start tracking the interruptions instead, by recording each one on a sticky note.

We opted for a sprint based approach in order to give ourselves a reasonably short timeframe, and immediately set about planning our first sprint. We guessed (yes, it would be wrong to call it anything but a guess!) that we'd lose about 50% of our time and capacity to unplanned interruptions, and also guessed at how much backlog work we could also get through. We got ourselves a board and started adding our story cards. It felt like scrum, it looked like scrum, but somehow, it wasn't really scrum.

Points Not Hours

In another tip-of-the-hat to Scrum, we decided to start estimating in points rather than hours. We had a well-established "1 point" item, which everyone was familiar with, and sized all our other stories relative to it. We took stories from our backlog, and sometimes needed to break them down into smaller stories, but we stayed away from using hour-based tasks.

I wanted to avoid using hours at all costs, for many reasons. Firstly, hours are an absolute value, while points are a relative value. Humans just happen to be far more accurate when measuring things using a relative scale than an absolute scale. Secondly I wanted the team to stop focusing on how long something would take, and consider other aspects such as complexity and unknowns with equal weighting. In my experience, as soon as you start dealing in hours, you automatically start focusing on how long things will take, and the other aspects get far less weighting.

Thirdly, I wanted to use something more abstract than hours, because it can often be too confusing for people outside of the team to understand why the total hours don't add up to the total available resource hours. I was hoping to avoid that particular political hot potato.

Tackling the Interruptions

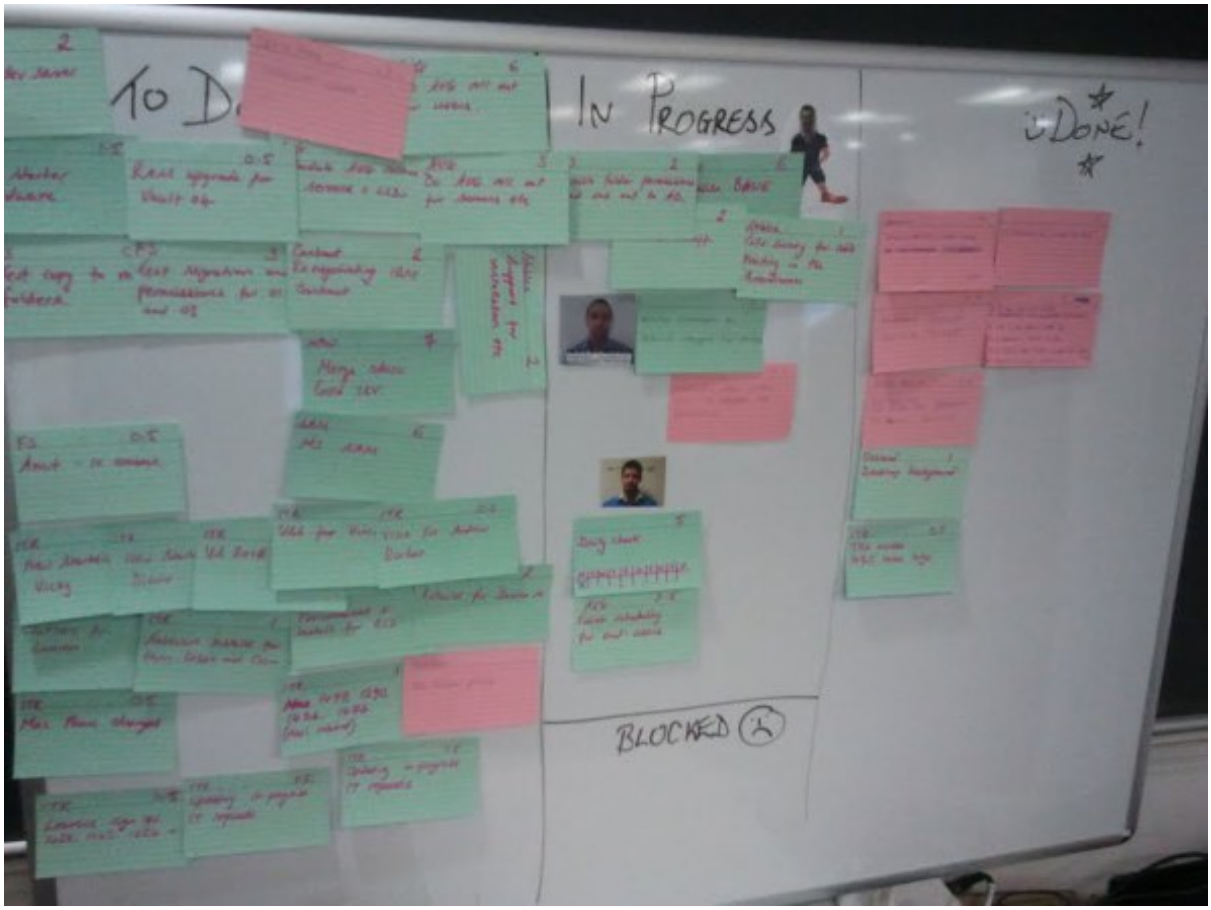
We knew we had an issue with interruptions, and there was a feeling that they were being prioritised over our backlog work, sometimes without justification. I suspected that in reality there were a combination of reasons why the interruptions got done over project work:

- They were usually smaller tasks and therefore easier to do!
- Someone was usually standing over you, telling you it was urgent
- They were "quick wins" which provided instant feedback, giving us the feeling that we'd actually done something useful!
- Sometimes it's not very easy to tell someone that you aren't going to do their request because it's not high enough priority

By writing the interruptions down on a card, and being able to look at the board and see a list of tasks we have committed to deliver, we were able to make better judgement calls on the priority, and also make it easier to show people that their requests were competing with a large number of other tasks which we'd promised to get done.

And so the first sprint began. Every time we got an interruption we wrote it down on a sticky note, and compared it with all of the cards in our "In Progress" column. If we felt it was a higher priority, then it

would go straight into "In Progress", replacing one of the existing cards (which would move back in to "To Do"). Incidentally (and this was raised in our first retrospective), the sticky notes turned out to be less sticky than the name would suggest, and kept falling off the board, so we switched to pink/red record cards and good old blu-tack instead)



The IT Ops "Scrum" Board

If the interruption was not deemed to be higher priority, then it either went into "To Do" or into an interruptions section of the Backlog. We sized the interruptions as we went along, always using the relative sizing we'd used for all the other stories.

Sustainable Velocity

One of the big desires of the team was to be able to give the business more accurate estimations on when our projects would be delivered. These were the bigger items on the backlog which we never seemed to be able to deliver on time because of all the interruptions.

We started out by having regular meetings with the business representatives, and asking them to arrange the projects in priority order. Once they'd done this we started to plan them into our sprints. After about 4 sprints we'd established that we were able to chomp through roughly 50 points of backlog work each sprint. After we estimated each backlog project, and armed with the knowledge that 50 was our magic number, we were able to give estimated delivery dates to the business backed up with figures to prove we could do it. Furthermore, we were able to push back on the business if they demanded that something get done by an unachievable date – we were able to provide them with the figures to prove that it couldn't be done!

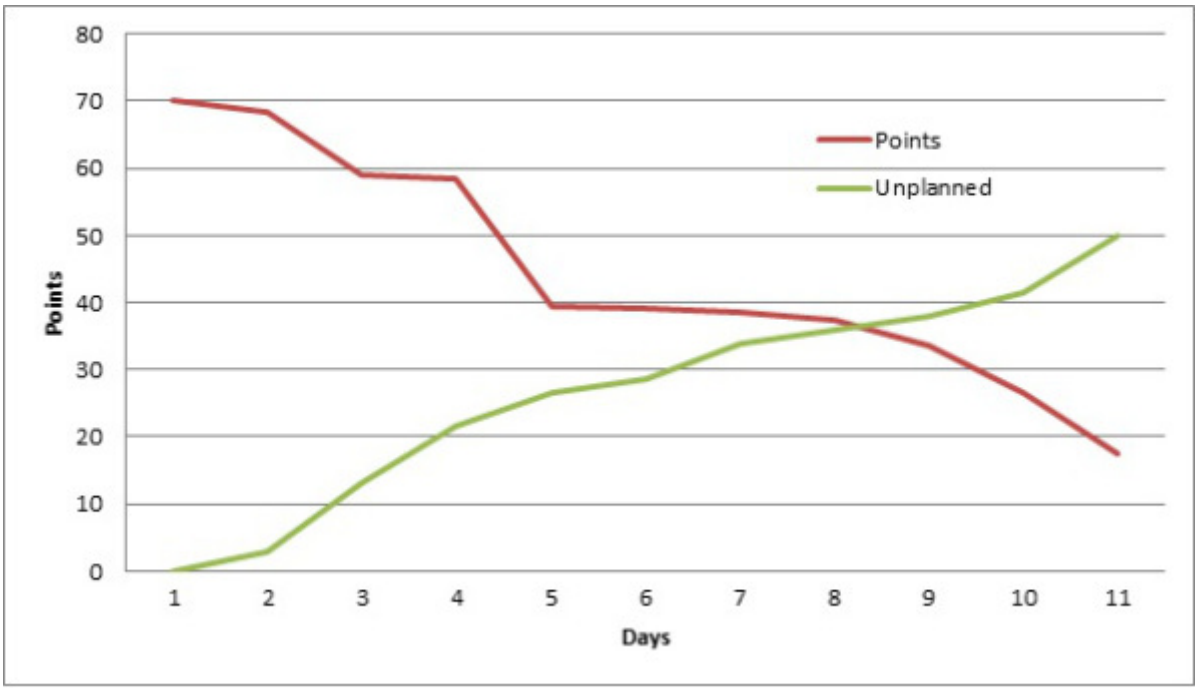
In order for our projected delivery dates to be reliable, we needed to establish a maintainable velocity. It's no good delivering 60 points in one sprint but only 10 the next, and so on – our velocity would be totally unreliable. To our surprise, the numbers were strangely consistent, but inconsistent at the same time. The total number of points we delivered seemed to vary between 90 and 120, but the number of backlog points (i.e. planned points) seemed to remain consistent at 50 points! This was excellent news for us, we only needed to be reliable on the backlog points in order to be able to give estimated delivery dates for our projects. But I couldn't help but wonder, "How are we able to do such varying numbers of points each sprint?".

Context Switching

If we added up all the time that we spent on unplanned interruptions and the time we spent on our backlog work, we would come up with a number significantly lower than the total number of hours available to the team. We were aware of this, but we weren't sure where we were losing this time. I had previously read a book called Slack, by Tom Demarco, in which he discusses the impact of context-switching on a person's productivity. Basically, context-switching between a number of different small tasks kills your productivity. In essence, we need to do less in order to do more. I wondered if it was this context switching that was responsible for our lost time, and if it was in some way responsible for the different total points we were getting each sprint. The evidence seemed to back up this theory. The sprints which achieved the most points tended to have a larger number of medium sized interruptions, whereas the sprints which achieved the lowest points were the ones with a mixture of large sized backlog stories, and a high number of small sized interruptions. The cost of being constantly interrupted was high – it appeared that being regularly interrupted with small tasks was costing us up to 20 points per sprint.

Burning Things Down

No, I'm not talking about arson, I'm talking about another tool we borrowed from Scrum; the burn-down. But in a slight twist, we overlaid it with the number of unplanned interruptions we were completing as well, so that we could get a feel for where our efforts were being spent each day.



Our First Burn-down!

Over time we noticed some trends – for instance, whenever we saw a sharp increase in interruptions, it correlated with a plateau on the burndown. This was completely in line with expectations, it just meant we were too busy working on interruptions to make any progress with the planned work. We analysed the burndowns in our retrospective and it helped us to visualise the impact of interruptions on our commitments.

Did It Work?

Both the IT Ops and the Infrastructure teams continue to use this Scrum-based approach. Both have made some more tweaks over time, and the process will continue to evolve. The experiment was certainly a success, both within the teams and within the business as a whole. The teams found this scrum based way of working to be refreshing, and the business enjoyed greater visibility of their projects, as well as having more reliable project delivery dates! The scrum approach also helped the Infrastructure Department work in a way that was more aligned with the development teams. They could align their sprints, help out with planning and estimation, and generally feel far more in sync with the development teams than previously. The main lesson we learned from this system though, was nothing profound like "doing less means doing more" or anything like that, it was simply that if you raise a card for every interruption then a standard sized sprint board just isn't going to cut it - you're going to need a bigger board!

About the contributor

James Betteley writes: *"Many years ago, someone decided to put me in charge of doing builds and releases of software. I can't remember why, I think everyone else was out at lunch. I inherited a system of batch files, shell scripts and all sorts of weird and wonderful things which, to this very day, I still don't understand. Since then I've made it my job to debunk complicated software systems, automate as much as possible and just try to make things more sensible. My main driver is bringing real business value*

through great Development & Operations processes, whether that be by using tools or implementing a new process or culture. Check out my blog at http://devopsnet.com/"

James Betteley

DevOps in the mix - John Clapham

John on Twitter: @johnC_bristol[104] - John's blog: johnclapham.wordpress.com[105]

MixRadio[106]

Timeline: Spring 2010 to Summer 2014

MixRadio is like a personal radio station with an uncanny knack for playing your favorite tracks. Available on the web and mobile, it allows free streaming of playlists without subscriptions or ads and is available in 31 countries including the US, Brazil, India and China. It has a catalogue of over 30 million songs. MixRadio grew out of one of the first online music services, On Demand Distribution (OD2), which was funded in part by Peter Gabriel. Based in central Bristol, the organisation has a reputation for forward thinking in both technology and ways of working.

Introduction

On a crisp, bright day in 2011 I took to the stage for my first ever public talk: *"Dev+Ops+Org how we're including (almost) everyone"*. I was nervous, but proud to represent the combined efforts of MixRadio's engineering organisation. The title was perhaps more telling than we first realised. We included people by default, just one of those agile principles that happens naturally - it would help us along, it would be fun. In truth, if we hadn't extended collaboration beyond engineering we may never have built such a strong Continuous Delivery capability.

An Unsustainable Pace

Our Continuous Delivery and DevOps journey began about a year before. At the time we were part of an autonomous business unit of about 200 people within Nokia Entertainment, based in Bristol and predominantly building music services. We used a mix of open and closed source technology, we were moving to a service oriented architecture, and we employed agile methods to get things done. The development teams averaged a server side release every three months. While that delivery frequency wasn't great, we were incurring a lot of costs that caused us to carefully consider our approach:

- Time. An awful lot of activity went into each release and it's related artifacts; scoping, release planning, testing, rehearsals, documentation. Much of it was bespoke and specific to that release, adding no lasting value to the organisation or customers.
- Cash. People worked extra hours, environments existed just to support release activities and quality gates, and we had a stack of pizza boxes high enough to reach the elevator ceiling. We were burning cash.
- Engagement. Some people found high pressure releases a buzz, but those pizza fuelled nights weren't sustainable. It was frustrating that those high odds were of our own making. Most people became engineers to make stuff and solve hard problems, not to babysit changes.

[104] https://twitter.com/johnC_bristol
[105] http://johnclapham.wordpress.com/
[106] http://www.mixrad.io/gb/en

- Reputation. Pressure, boredom, late nights and forced decisions aren't ingredients for a high quality product, and defects often cropped up at the worst time. The mitigation for this was often more process and downstream checking, adding to cost and delays.
- Opportunity. Customers had to wait for a release before they saw features, potentially costing us competitive advantage. It created a lengthy product feedback cycle, hampering our ability assess and react to user behaviour. Mirrored on the technical side, three month change cycles reduced our opportunity to evolve our architecture and technical approach.
- Product Flow Interruptus. Perhaps most seriously, product development stopped during the release phase. Engineers were either involved in release activities, or last minute fixes. Fear of missing precious release opportunities influenced product decision making. The cost was greater than the total number of mythical man days of the release. Flow was interrupted, people took time to recover and pick up their threads.

Reviewing the above list you might notice a theme - risk. Attitude to risk was driving behaviour, and it manifested in much of the process. Not wanting to risk downtime, Operations checked everything thoroughly and requested documentation to support unfamiliar applications. Not wanting to risk errors in production, staging and pre-live environments were created. Not wanting to risk errors in pre-production, quality assurance teams were established with multiple handovers. Our system bore many of the hallmarks of Risk Management Theatre[107].

A Pause For Thought

It was clear the price of release was too high, and the simple metric of talented people getting frustrated reinforced that message. You might wonder how we'd got into that situation, and why things hadn't changed sooner. In fact it evolved slowly, as it is hard to view a whole organisation in depth. Issues aren't always obvious as they are obscured by local budgets, politics, silos and shifting fiefdoms. Often the impact of a problem is not felt in the area responsible for introducing it. Few people are in a situation to take this holistic view, and may even consider it not in their interest to comment. The crucial point is that most teams were working to best practices, within their own areas. These local optimisations meant that the overall product delivery system suffered.

This situation couldn't continue, but it is hard to break out of this kind of deadlock. There was a long period between knowing what to do at a technical level, and actually starting. During that time there were many courageous conversations and tough decisions, but the end result was sponsorship from the leadership level. This was a crucial and foresighted piece of leadership - investing some of the team's capacity in improving capability, rather than building features and running systems.

The first smart move was to pause and think, to really zoom out and consider the problem. It is comforting to make improvements on existing process and tools, but the danger is iterating on an area that won't reward investment. This is a common theme, by way of example consider the Blockbuster example[108] - improving a product without realising it has already been disrupted.

To build Continuous Delivery tools a Scrum team was formed. In creating a new team there was every risk of contributing to the problem - another interface, and other step between concept and customer, or as we soon termed it "Laptop to Live". Our team was different to the much debated DevOps Team approach[109], as it's remit was to deliver a product to Research & Development and then disband.

[107] http://continuousdelivery.com/2013/08/risk-management-theatre/
[108] http://www.bbc.co.uk/news/business-21047652
[109] http://continuousdelivery.com/2012/10/theres-no-such-thing-as-a-devops-team

To start with, the focus was tools. It soon became clear that tools weren't the only constraint, and we began to understand the close relationship between tech and ways of working. Each time we attempted to simplify tooling, we would hit a process or people constraint. Here is a typical (but not real) requirements gathering conversation:

> "This environment appears to be a bottle neck" "You can't remove that environment because its needed for performance testing" "Why is performance testing needed?" "Performance tester says so" "Why?" "It's best practice and…"

And so on. So it became clear that both tooling and ways of working were closely coupled, and we'd need to do more than just deploy a pipeline. There were touch points on the software throughout the organisation, understanding those, and their true value would be crucial to success.

First steps

The amount of work that went into building our Continuous Delivery capability can't be done justice here, and it was by no means a smooth path. On reflection, we were able to gather some of the initiatives into the areas below although at the time our approach was far less structured.

- Improve, continuously. Learning behaviours, and a desire to learn are the catalysts for this kind of change. It was necessary to iterate quickly, learning what to change and build and what to leave alone for now. This was especially important as organisational and process changes often set up the technical challenges.
- Listen, constantly. We used formal exercises like value stream mapping, and agile workshops, and tried to find a sensible balance between iterating in the right direction and starting absolutely every thing again from scratch. It was also important to listen to dissenters, often uncovering areas we hadn't considered and new improvements.
- Establish a shared purpose. We didn't realise the significance of this until much later, but our sound bytes like "Laptop to Live" were actually useful rallying points that encouraged people to aim for the same goals. Other methods included sharing and showing progress through demos, monitors, and taking a community approach.
- Establish principles. There were a few key phrases, or principles that crept into the vocabulary of engineering - "never break your consumer", "put responsibility in the right place". Some were reminders that increased trust and increased freedom introduced the possibility of going wildly off track. These basics allowed freedom, and invited technical and non-technical solutions to problems.
- Judicious automation. A common slogan is "Automate Everything", but we tried to make automation targets go away. Once something is automated it becomes an asset requiring maintenance, it becomes less malleable and actually impedes ability to change. Where possible we used simple systems, like a toy badger as the token to lock the build transaction.
- Earn and build trust. Trust is commonly cited as a foundation for DevOps and Continuous Delivery, but it doesn't just happen. Management can't declare "we all trust each other now, get out there and do trusting". Trust has to be earned repeatedly, otherwise it evaporates quickly. Keeping progress visible, inviting feedback and showing successes all helped here.
- Design the organisation for flow. Some groups are more beholden to hierarchy than others. We found it useful to avoid dividing teams with separate managers, styles and incentives. This reinforced our joint purpose, encouraged collaboration and sped communications.
- Build damn good software. Do I need to say more?

Momentum

Over time a new perspective on risk emerged, a pragmatic approach where the quantity of process and checks around a change were proportional to the risk in carried. Standard changes allowed changes within than category to move with minimum attention. To start with that assessment was carried out by Development and Operations together, but after a learning period developers categorised their own changes.

After a relatively short period of time, multiple daily releases became routine. I enjoy bragging about daily release count as much as anyone, but the real milestone was the ability to release often enough to match the pace of development team throughput.

As intended the Continuous Delivery team disbanded, and the MixRadio engineering community as a whole took forward the release toolchain and furthered our DevOps ways of working. Momentum was maintained, Continuous Delivery stayed in the limelight, and we found a few nice ways to promote internally. One example was celebrity deployments by visitors, including Nokia CEO Stephen Elop.

The tooling continued to improve, albeit slowly, on a best efforts basis. Improved tooling revealed more process and ways of working issues that needed attention. Increased confidence enabled bold decisions like removing pre-production environments. With each change quality was monitored, and invariably improved.

A Tipping Point

The MixRadio organisation is a hungry one, keen to improve and better itself. That meant just like the environments we'd pruned out the Continuous Delivery tool itself was subject to scrutiny. A welcome change in hosting strategy permitted migration to Amazon Web Services. The CD tool could be adapted to deploy to the cloud, but it was coupled to the processes it modelled and reflected the conservative organisation in which it was brought up. The architecture had evolved, with fine-grained services reducing queues and making it less likely that changes would conflict. Roles had changed and knowledge had been developed over the years. The first pipeline tool helped engineers to deploy, now engineers had good experience of production systems. Operations needed to turn their attention to specialist areas including infrastructure, security, and provisioning. All these factors were positive, but it was clear the pipeline was on the verge of limiting our delivery capability.

A new Continuous Delivery tool was to be built, but this time things preceded very differently, a mark of the organisation's maturity and learning capacity. It didn't take long to gain sponsorship for a new team, and this time it was an autonomous group of engineers. Organisational buy-in was a given and this enabled a firm focus on technical challenges. The new system took full advantage of the changed environment, organisation and architecture. As opposed to being driven by the need to protect from failure and bolt down every risk, the new system relies on trust, simplicity and mastery. There are just two environments, a handful of commands to deploy, no tooling orchestration, and no exclusive locks.

Reflections

Continuous Delivery has brought many benefits to MixRadio, improving speed of execution and efficiency. Although highly subjective, engagement and motivation appear to have improved as people focus more on problems that matter and are able to take full responsibility for their code. Having a single, simple, visible goal was crucial for bringing change. In some cases this united teams and encouraged

collaboration. In other cases change was less welcome. What mattered was the sense of purpose it brought and that conversations were started. A sense of community helped the changes in many ways, from building trust to gaining feedback and understanding the problem and goals.

While trying to improve delivery capability there appeared to be five interdependent factors: tools, culture, architecture, organisation and process. Often one would enable another, and when improvements were made a new constraint would surface elsewhere. Culture has a significant impact on technical choice, lightweight approaches become viable in environments of sufficient mastery, trust and autonomy.

From this it seems important to view technology and process as learning tools, to not be sentimental and refactor both mercilessly. Unless your business or competitors stand still they are never done.

About the contributor

John Clapham is an Agile Coach at Kainos, digital solution providers for a growing number of high profile government, commerical and public sector projects. Prior to his current role John was a Software Development Manager at MixRadio, based in Bristol. The role included a sojourn as Product Owner for the Continuous Delivery Team, leading to a strong interest in DevOps. John had a dubious start in IT, it began with several weeks erasing magnetic tape reels with a huge magnet in the staff canteen. Since then John has built web and enterprise apps in the publishing, teleco, commerce, defense and public sector arenas. These days his focus is firmly on coaching teams, and creating an environment that is both productive and enjoyable to work in. This has lead to forays into Scrum, Kanban, Lean, DevOps and just about everything in between.

John Clapham

You write it, you support it - Jennifer Smith

Jen on Twitter: @JenniferSmithCo[110] - Jennifer's blog: jennifersmith.co.uk[111]

ThoughtWorks[112]

Timeline: October 2009 to present (October 2014)

ThoughtWorks is a global company of passionate technologists. They specialize in disruptive thinking, cutting-edge technology and have a hard-line focus on delivery. Their clients are ambitious, with missions to change their industry, their government, and their society. They provide software design and delivery, pioneering tools and consulting to help them succeed.

Understanding infrastructure

I recently got to spend a few months working with a client's infrastructure and sysadmin team. With the help of my very patient colleagues, I learn about many new and unfamiliar concepts: networking and DNS, database cluster management, debugging process system calls, among many others.

In addition to supporting various pieces of infrastructure, we were responsible for production support of the services and applications produced by the development team. For example, the response rates on our customer-facing site dropped significantly one morning and set off an alert. Our job was to diagnose and stabilise the issue and finally conduct more investigation into the problem and figure out how it could be avoided in future.

I had transitioned from writing software to supporting that very same software in production. With this change of role, I noticed there were a some common themes in the problems that we were experiencing day to day.

- **Excessive logging**: Filling up the log files with unhelpful messages or expected exception messages. With a high signal-to-noise ratio in our logs, we found it difficult to home in on the messages relevant to the current problem.
- **Insufficient logging**: Conversely, we had a problem with too little logging. Swallowed exceptions and error messages that would have been useful to see in the logs, batch jobs that ran silently without telling us whether they had failed, completed or even started in the first place!
- **Coping with failure**: Applications failing to deal with Murphy's Law[113] - "Anything that can go wrong, will go wrong". Outages caused by upstream services and networks going down or slowing down, bad input from users. Sometimes our applications would fail in very surprising ways in these conditions: blocking threads, eating up memory and yet more logging!
- **Lack of feature degradation controls**: when parts of the system did break, we did not have controls to allow us to degrade elegantly. When the session database started misbehaving, we had to switch off the entire site to address an issue that only affected a small proportion of visitors.

[110] https://twitter.com/JenniferSmithCo
[111] http://www.jennifersmith.co.uk/
[112] http://www.thoughtworks.com/
[113] http://en.wikipedia.org/wiki/Murphy's_law

- **Lack of metrics and monitoring**: When all these problems start happening, we would often find out far too late. We often tended to find that we needed to add metrics and monitoring long after a new feature had been finished.
- **Unfamiliarity with production volumes**: in developing features we would often fail to take into account production data and access volumes. In one case, we developed a paged list of user records for our administrators. We had assumed that there were likely to be no more than a handful of rows per administrator. When we reached production with this feature, we noticed that there were a handful of administrators who had imported hundreds records, making our paged list completely unusable (and worse still, causing performance issues for other users of the site).

All of these categories tended to contribute to either causing production issues or just making them difficult to detect and diagnose.

We attempted to address these issues by attempting to fix things ourselves, raising bugs to the development team and trying to educate the developers on how to make new features more supportable in production.

This did something to address the individual symptoms, but not the underlying problem: we were developing software without sufficient consideration to how they would run or fail to run in their intended environments.

There was a big temptation to point the finger at our development team: to blame them for writing unsupportable systems. But what stopped me doing this was that I had been on that team but a few weeks before. I respected my former team-mates and I knew that we collectively cared about continuous improvement and the quality of our work.

So what was the problem?

Feedback loops

When you spend a lot of time working with software you reach the realisation that gaining and acting on feedback from your environment is an important skill to learn. Feedback comes from your customers influencing your features, your code changes passing or failing a test or your design choices making software easier or harder to change and modify.

Regardless of the nature of the environment from which you are receiving feedback, the shorter the feedback loop the better chance you have to quickly respond to changes. The ability to immediately see the outcome of your actions is a far more visceral experience, leading to more effective learning and improvement.

This manner of thinking underpins a great deal of practices we talk about: continuous delivery involves rapidly and continuously delivering software to its target environment, automated tests aim to give us immediate feedback to whether our code is doing the right thing, guerrilla user testing aims to quickly test elements of user experience before being realised in functioning software.

Feedback loops from running systems

The process of writing code, deploying it, then observing its behaviour in a production setting is another feedback loop. In our situation, the feedback loop from our running production system looked a little like this:

Production feedback loops

The development team would write code to be run in production. When something went wrong later on down the line, the infrastructure team was responsible for firefighting and raising issues back to the development team. This took up a great deal of our time and energy in diagnosing issues in unfamiliar code. It also took up a lot of the development team's time and energy in responding to issues that were often caused by changes made many weeks ago. On top of that, issues were affecting the users of our production systems: problems took a long time to diagnose and even longer to fix permanently.

How could we improve this situation for all parties involved?

Shortening the feedback loop: "You write it, you support it"

One way was to make the development teams writing the code responsible for supporting it in production. Place them on the front line of support for the systems they are building.

If a problem occurs on the site - for example response times or error rates suddenly increase - the people responding to the problem are the same group that wrote the software in the first place. Now this team will be the ones trawling through the log files, piecing together events, restoring service and then rectifying the problem or improving visibility and monitoring.

Without trickling these issues through the infrastructure team, we could start to reduce feedback times by reducing hand-off time and context switching and hopefully improve the time-to-fix.

There are some more rewarding yet subtle benefits of moving responsibility for production support. Supporting their work in production allowed our application development team to understand how to make their software more production-supportable. Experiencing first-hand the pain of unusually verbose and usually terse log messages and you are likely to think more about how you write logging code in the future. Spending a lot of time fixing a poorly performing SQL query slowing down page requests unexpectedly and you are likely to start considering production data volumes and constraints during development.

This direct visibility and awareness of the problem results in glaring issues being quickly addressed but also has an effect on future development. After a time, the development team will start to build more supportable applications as they learn what makes the job of supporting their software easier.

Collaboration

As we began to push out the production support to the development team, our involvement in production support as the infrastructure team did not completely diminish. Enabling our colleagues in the development team to support their own work necessarily involved a great deal of collaboration.

Shortened feedback loops and increased collaboration

As a set of experienced sysadmins and network engineers, my colleagues on the infrastructure brought a wealth of experience in debugging and diagnosing problems in running systems, experience of debugging tools, and fault localisation. Recasting my team as the guardians of production rather the custodians and as teachers rather than doers meant that we now had to work on sharing skills and expertise.

Trying this yourself

Many of the places I have worked would have benefited from moving production support to the application development teams. In fact, a growing number of organisations are already taking this approach. Having said that, there are a few important considerations you need to take into account before trying this in your own organisation.

Is it appropriate for your environment?

The ability for teams to support their own systems depends very much on the nature of these systems. In my case, we were working with self-hosted, web-based applications with rapid change cycles.

For long running, stable applications or 'shrink-wrapped' customer-configured applications, the gains you get from devolving support into the application teams may not be as great.

If this approach is not appropriate, there are other ways you can shorten that feedback loop. If you have a dedicated infrastructure or operations team, perhaps having your application development team members to do occasional rotations into this team might go some way to addressing the gap.

Plan for a bumpy start

If you make application teams responsible for production, there is going to be some initial pain. When we moved responsibility for application support, it did not immediately result in magically self-supporting and self-healing systems from day one. Nor did the application development teams immediately acquire the skills and expertise needed to diagnose and fix production issues.

Expect that issues will take a longer time to diagnose and fix initially. Over time you should expect the frequency and fix-time to decrease as developers gain the skills and make their work more production-supportable.

Humane out-of-hours support process

The process for out-of-hours support must be humane and reasonable. This is important regardless of which team is providing production support. If you require systems to function out-of-hours, invest the time in making issues recoverable and detectable early. Take interrupting a night's sleep or an evening with the family as seriously as the impact to customers or end users.

If you are not already doing so, log and measure the number of production support issues a week, conduct post-incident reviews and actively work to reduce the frequency and time-to-fix for incidents.

Keep getting better

Simply moving the responsibility of production support to your application development teams is not enough. Your infrastructure team will be freed up to work on what's important to them, but now your application team will stop having time to develop business critical features. In fact, that is a common fear and objection to this change of responsibility.

To address this you need a dedication to continuous improvement: a commitment to making production systems more robust, recoverable, and inspectable.

Self-organising and empowered teams

At the foundation of all of these things there has to be a self-organising and empowered team. Your team needs to have the ability to influence their own destiny, work on the things that are important, instigate change, and make improvements. Disclaimers and caveats aside, "You write it, you support it" is an important step on the path to building more robust systems by shortening feedback loops and increasing visibility of issues. Try it out!

Credits

Cartoons by Greg Skinner Adapted and extended from a blog post[114].

About the contributor

Jennifer Smith is a Software Consultant for ThoughtWorks Australia. She originally got into software development through the coincidence of mandatory C++ classes that formed part of her Music Technology degree. She equally accidentally became interested world of infrastructure and operations worked â€˜cos you gotta get software out somehow! Grudgingly this has translated into actually enjoying learning more about infrastructure and understanding how to keep things running in production.

Jennifer Smith

[114] http://www.jennifersmith.co.uk/blog/2012/10/06/you-write-it/

Continuous Delivery across Time Zones and Cultures - Sriram Narayanan

Sriram on Twitter: @sriramnrn[115] - Sriram's blog: sriramnarayanan.com[116]

ThoughtWorks[117]

Timeline: July 2011 to June 2012

ThoughtWorks is a global company of passionate technologists. They specialize in disruptive thinking, cutting-edge technology and have a hard-line focus on delivery. Their clients are ambitious, with missions to change their industry, their government, and their society. They provide software design and delivery, pioneering tools and consulting to help them succeed.

Introduction

Aiming for Continuous Delivery involves a journey of people, of processes, and of tools. It involves letting go of and taking up of responsibilities, changing how people are measured, and resolving all manner of issues. Everyone needs to be a part of the solution and make things happen.

Over more than 6 years ThoughtWorks has helped one of its clients on their journey of broadening their customer base for its online ticketing services. The application stack comprises an ASP.NET based web app that supports theming for multi-tenant hosting, a .NET based web service layer, various custom components developed in house and sourced from third parties, and an Oracle database. The web services also integrate with multiple third parties for ticket reservations, bookings, and validations of various sorts.

The Origin of the Build and Release Team

On smaller teams, the concept of shared responsibility works well. When the same source repository and environments are shared across time zones as well as organisations, there is a need for a single group to hold the pieces together. The programme had a Build and Release team which maintained build/deployment scripts and Thoughtworks Go-based pipelines.

Various programme managers tried to rotate members in the team, which meant a number of developers got exposure to some form of infrastructure. However, when you have only developers working on infrastructure things may sometimes be suboptimal. Furthermore, the developers sensed they were getting out of touch with the application domain and wanted to get back to development. Finally, the team had grown weary of dealing with various instabilities - even with admin access there was only so much they could fix.

[115] https://twitter.com/sriramnrn
[116] http://www.sriramnarayanan.com/
[117] http://www.thoughtworks.com

Taking stock

We invested time in learning how the build scripts were organized, how and why we branched, and the reasons for build and test failures. Certain jobs would run correctly on only one particular VM or a set of VMs; an integration test would fail intermittently and then pass soon afterwards. Despite parallelised builds of various components across VMs, we didn't have predictable timings for the overall build stage. Our regression testing would take up to 18 hours, and various components would take a much longer time to build than other components. Furthermore, during user acceptance testing (UAT), users would suddenly experience an unresponsive website.

Performant environments and reduced functional test times

The top complaint was "environment issues", which were a combination of deployment misconfigurations and overcrowding of VMs. Regression testing took 18 hours on a good day if there were no environmental issues. We therefore decided to stabilise the regression testing environment and speed up regression testing, followed by stabilising the UAT environment, and lastly, stabilise and speed up the build and packaging process.

By dedicating high end VMWare servers for regression testing on our Selenium grid we brought down regression testing times from 18 hours to 2 hours. Next we dedicated three high end servers and SAN storage for hosting just the UAT environment and ensured it was not starved of CPU resources. Later, we isolated another physical server for performance testing. In all these cases we appointed individuals to be complete administrators of a specific environment, and introduced low-level monitoring of physical and virtual infrastructure. Our intent was to delegate administration, to remove the fear of infrastructure, to get people exposure across the full stack, and help them understand the infrastructure was now capable of handling load.

Network issues, bandwidth issues, and network latencies

We often had embarrassing situations where the application failed during showcases, where code checkout took too long, and/or functional tests intermittently failed. While infrastructure can be scripted, the network layer is often mysterious and you have to be creative - you can only improve latency to an extent across distance. We brought in a lot of network layer fixes including redundant data links, SNMP monitoring of the link between the UK and Indian data centres, and ensuring everyone had a working SSL VPN connection. We also migrated the whole codebase from Subversion over *svn+https* to Git over *git+ssh*, with local mirrors reducing latency and SSH access improving speed.

Reducing build times

The core components in the application stack were compiled in parallel across a farm of build agents, and there was often no predictability in the times it took for various application components to compile. We used the Go API[118] to get a sample of the build times for various parallel jobs, and split the list of jobs into two – long running and short running jobs - for specific build agents. This brought the compile phase down from 75 mins to 19 minutes.

[118] http://www.go.cd/documentation/user/current/api/index.html

During this analysis we also noticed that several components were getting compiled as part of other components' build jobs. We created a team of two developers who broke down the 39 job compile stage into two smaller stages – one of independent components and DB initialisation, followed by another stage that built components with dependencies. This meant issues with independent components or database initialisation could be reported faster.

Providing predictable environments

The existing VMs had been in existence for 5 years and had become virtualised Snowflake Servers[119]. Our goal was to understand what it took to get a generic build agent. We'd create one, run a variety of build jobs on it, understand what was missing, and then spin up a new build agent and test that out. Over a period of time, we understood what it took to create a generic build and used Chef based cookbooks to create the constituent VMs of a UAT environment. We moved to a forced refresh of VMs every 6 weeks, and created a Go job that could trigger envronment creation on demand.

Improving test predictability and stabilizing tests

Over a course of time, we identified several reasons for test failure. Parallel tests used the same database, there was no proper test data management, Selenium timeouts were not correctly configured, and our Selenium version was old. All of these caused intermittent problems, which people would overcome by simply retriggering the build. Once predictable test environment was in place, the functional test automation team improved Selenium timeouts and we migrated to Web Driver.

Monitoring builds

We set up Graphite monitoring to measure the times that individual jobs took, and to notify us if a job took longer than usual. To help everyone realize that we were now having more green builds, we started to produce a daily report of the green and red builds each day.

Antipatterns

We had to deal with the following 'antipattern' behaviour from developers:

1. Checking new code onto broken code.
2. Retriggering a broken build without actually fixing the core problems.
3. Expecting the Build and Release team to investigate broken builds.
4. Call for email based notifications when the build breaks.

Early on the teams had a low confidence in the build and test environment, choosing instead to verify a build independently in a team-specific environment. People would commit code even if a build was broken, retrigger various jobs to get a green build, and ask the Build and Release team to investigate broken builds for them. My colleague and I realized that everyone had stopped treating the CI system as a CI system, and considered it merely a compilation farm.

[119] http://martinfowler.com/bliki/SnowflakeServer.html

Red (broken) build stages

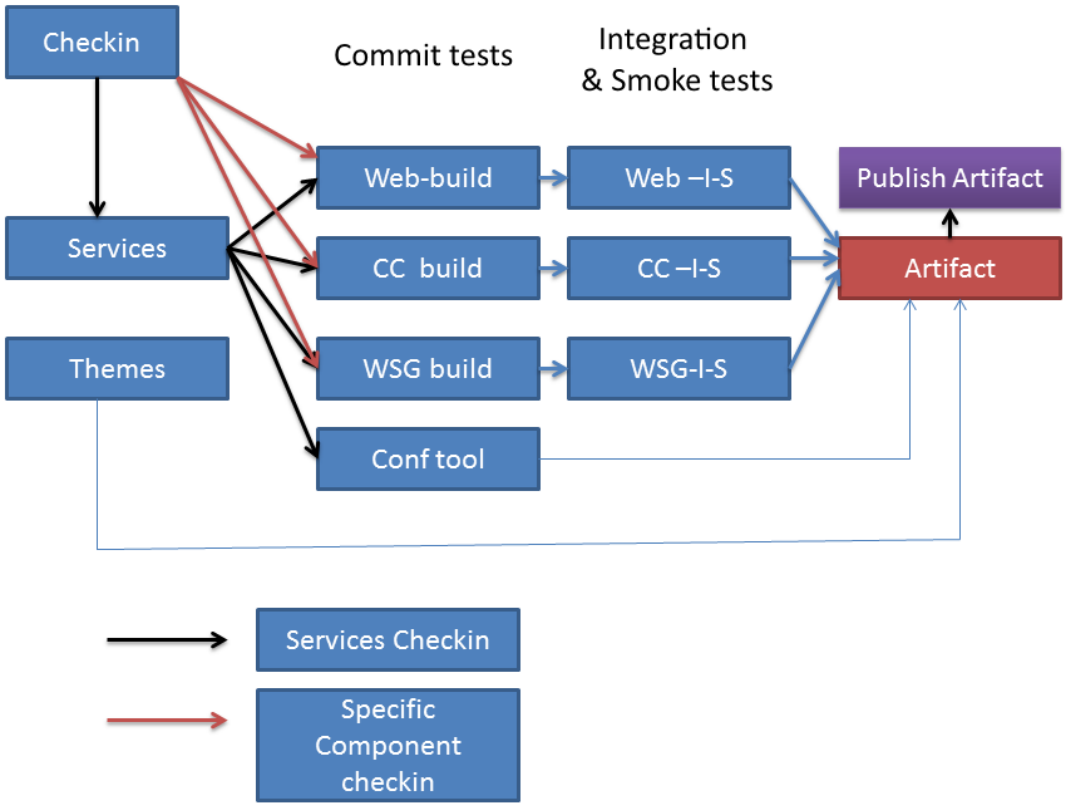
Being a purist I disliked the notion of commit tokens and didn't want to send out email notifications. Instituting such measures would introduce further bad behaviour and move us away from the practice of Continuous integration. Instead, we introduced pre-commit hooks that would only allow commits on green or if the message contained "[Fixing Build]". This didn't encounter any opposition and managers liked the onus upon fixing the build.

We sometimes took shortcuts to get a green build, but we actively worked with developers on the longer term fixes. Once we had build monitoring in place it was easy to point out the top 5 jobs that failed the most and required attention. Based on this the programme manager would allocate developers to look at those, and most errors were caused by dependency management rather than environments.

The pipeline structure

We ended up with the following pipeline:

Some artifacts were built in India and then published to the UK where they were integrated with the artifacts built by UK-based teams:



Lessons we learned for effective Continuous Delivery

Technical

1. **Push as much into the VM Template as possible** By pushing the platform stack binaries onto a template, we used Biztalk and SQL Server configuration tools to reduce VM preparation from 45 minutes to 10 minutes.
2. **Make use of OS facilities where possible** To run our Chef scripts on startup we wrote computer startup scripts instead of user logon scripts
3. **Get system administrators comfortable with configuration and infrastructure** Had we been comfortable with configuration management and automated infrastructure sooner, we could have reduced cycle time sooner.

Tactical

1. **Prioritise your solutions** While there were many burning fires, we chose to stabilize critical elements first.
2. **Facilitate rapid identification of failures** The first investigation of a broken build should be by the development teams so they own the act of keeping the build green.
3. **Showcase your ideas rather than seek permission** We spent a lot of time caught up in permission seeking calls to use Chef, to discuss revisiting build scripts, etc. Showcasing ideas requires time, and it is best to always make sure you have a proof of concept before you start talking to the business.

Personal and team-related

1. **Ensure excellent relations with counterparts on the other side** Because we worked well with our counterparts on the UK-based build and IT support teams we got a lot done over email or the phone rather than via formal calls or inter-organisation catch up sessions.
2. **Ensure regular team member rotation** Exposure to infrastructure helps developers consider deployment and maintenance scenarios during development, and keeping the same people on a team for a long time can lead to them becoming out of touch or bored. Rotation can be a win-win.
3. **Work on getting the staffing right** Since our team had a development background we were able to work across the full stack and resolve issues earlier. Later on, we enabled developers to understand infrastructure and they were able deal with the full stack.

Organisational

1. **Have a 'one team, one company' mindset** Team and organisation affiliation leads to a 'Not Invented Here' syndrome, treating 'offshore' folks as 'resources', and leading to a general slowdown. Thanks to a programme level drive to have a 'one company' mindset, we knew we were in this together. This was the single biggest contributor to success.

Remaining challenges

There are some other important activities we didn't get around to solving:

1. Test data management Managing test data effectively is highly desirable in integration and functional testing. Conventional setup-teardown of test data may not work well if the tests are run in parallel.
2. Test data at volume We used to regularly face issues due to out-of-date Production data, but newer data was inaccessible to us due to data privacy issues. Somehow the technical team's arguments for data obfuscation didn't make it through.
3. A single monitoring dashboard We had separate dashboards for virtualisation, storage, networks, builds, and tests, as well as custom monitoring. We planned on having a unified dashboard, but didn't get there.
4. Environments on demand We looked at a number of options for allowing development teams to create and manage environments on-demand, but didn't have time to select an approach.

Build and Release, or Infrastructure Engineering?

While the team was chartered with taking care of build and release, we focussed solely on infrastructure automation, monitoring and predictability. Once we brought these in, developers and QAs were able to ensure test stability. Similarly, developers were able to work on packaging and deployment with more confidence than before. While we were lucky that developers picked this up, this also highlights that packaging and deployment is best handled by developers, since they know how the application works and what configuration it needs.

About the contributor

Sriram Narayanan *is a Problem Solver, Troubleshooter, Technology Enthusiast, Musician. He helps organisations on their Agile Transformation journeys. He loves Release Engineering and Infrastructure Automation. When not working on reviving Belenix (www.belenix.org), an Illumos based distro, Sriram is busy learning new things like swimming and dancing, or on the air as AG6WQ.*

Sriram Narayanan

DevOps in an Enterprise environment - Jan-Joost Bouwman

Jan-Joost on Twitter: @JanJoostBouwman[120]

ING[121]

Timeline: early 2012 to July 2014

Introduction

This is the story of the journey of ING Bank Netherlands from a process oriented to a content driven DevOps organisation.

Something about ING

ING is a global financial institution, with its roots and headquarters in the Netherlands. ING, consisting of ING Bank and NN Group, offers banking, investment, life insurance and retirement services. On the Banking side some 63.000 employees work for over 32 million private, corporate and institutional customers in over 40 countries in Europe, the Americas, Asia and Australia. This story is about the IT department of the Dutch Retail Bank and its journey towards DevOps. A department of over 1800 people, working both in development and application service management, currently with 180 DevOps teams working on some 2000 application services in several hundred value chains of different complexity.

Reasons leading to the transition

The reasons that led to the transition from a process oriented to a content driven DevOps organisation were those of most established banks in mature markets, especially those that grew through mergers and take overs in a time that the market was still expanding. In an expanding market with good profit margins there never was the urge to consolidate brands, departments and IT applications. ING (Internationale Nederlanden Groep) was no exception, being the result of a merger between privately led NMB Bank Nederland and state bank Postbank and later insurance company Nationale Nederlanden. The first merger was in 1989, with the merger with NN in 1992 but for years the two banking brands, with their separate IT landscape existed next to each other with little interaction between the two. When the notion struck that the market was levelling off it was time to look at cost. Both the cost/income ratio and the percentage of maintenance cost of the total were on the high side. One of the first things to do was to finally merge the brands, not just in the high street, but more importantly in the data center. A ‘target end landscape’ of applications was chosen, and a massive data migration was started to merge the two datasets. This migration was done per client segment, because different segments use different applications. The final segment of business clients was migrated in Q3 2014.

Although this data migration programme will reduce cost greatly in the future, on the short term it didn’t. So alongside it a decommission programme was started in 2010. This programme looks at all applications

[120] https://twitter.com/JanJoostBouwman
[121] http://www.ing.nl/

to see which are not used anymore and where there is overlap in functionality. An applications that had its data migrated to the same application on the other side of the bank are the obvious candidates for decommissioning, but it also turned out that there were a lot of applications with just a few users, who only used it as a backup for another system. So far 738 applications have been decommissioned and a target of a total reduction of 50% of the current number of applications is set for when the programme will be concluded in 2016.

These two programmes helped reduce the maintenance versus new functionality ratio by reducing our IT footprint, but they did not solve the problem yet that any new functionality took months to reach production. For a typical application the time between the initiation of a project as an idea and the release into production was at least a year. Furthermore, when new functionality was finally released into production it often had quality issues or did not deliver the value that the business had hoped for. The complexity of our IT landscape played a major part in this, but also the way we developed with Waterfall contributed.

To sum it up: we were facing three challenges:

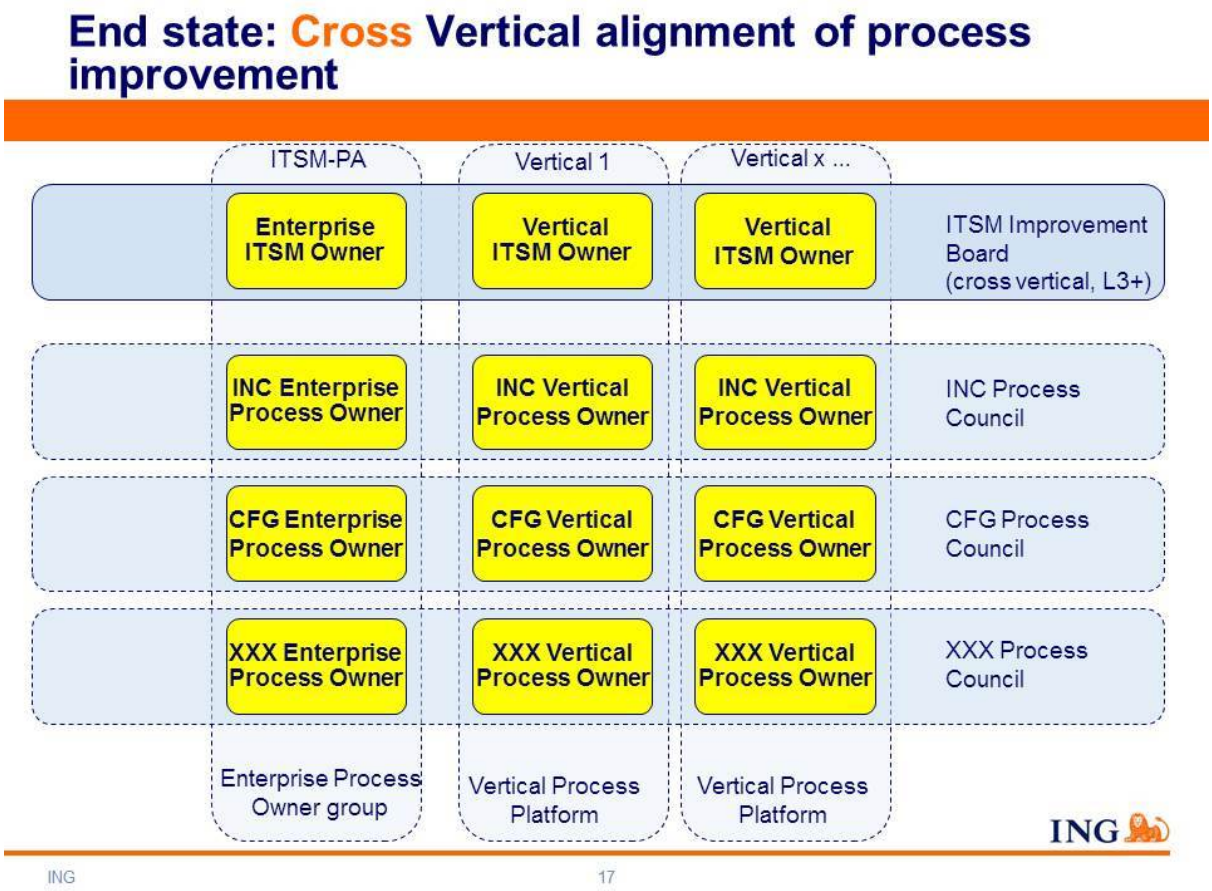
- the need to reduce cost
- the need to reduce time to market
- the need to improve our quality

We soon learned we were not unique in this.

The process organisation

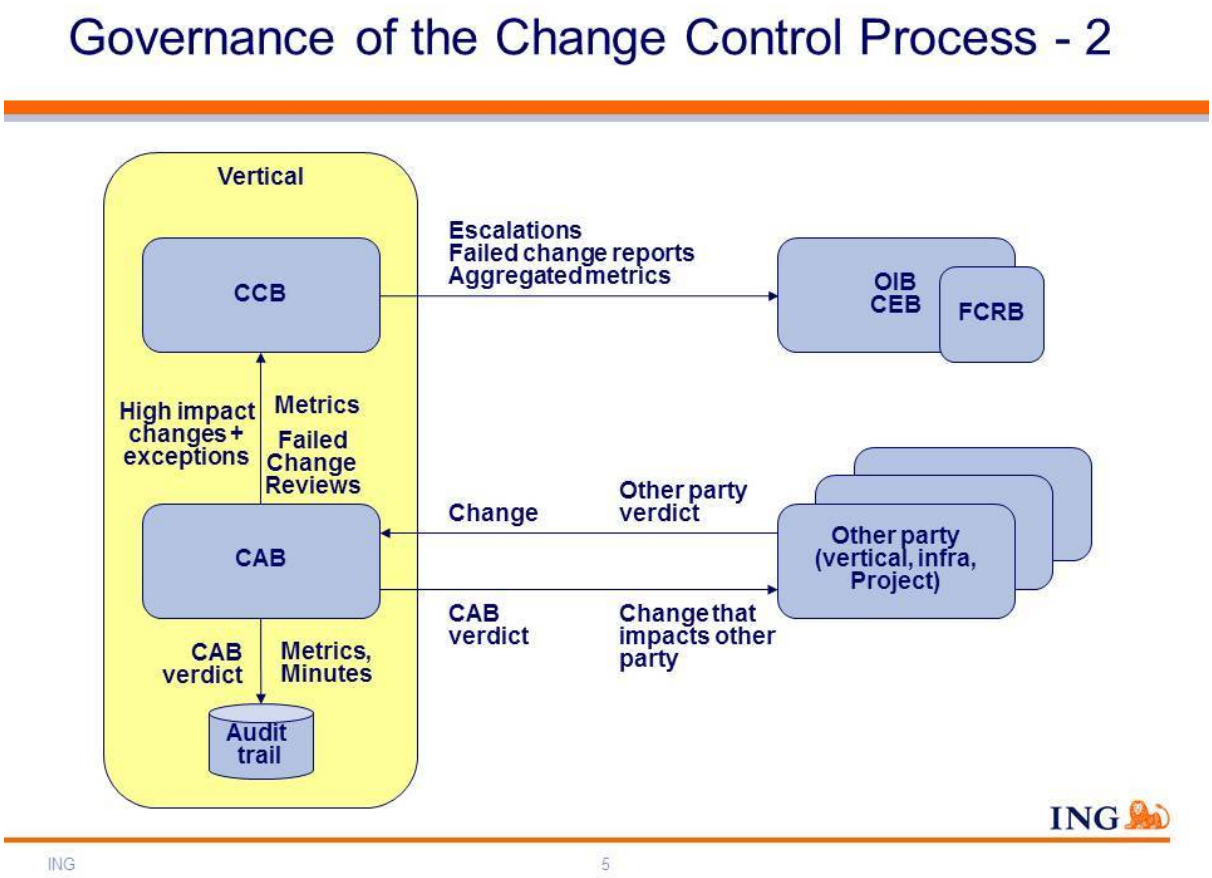
To fully understand where we were coming from I need to explain a little bit about the 'process organisation' we were in the past. For Service management it really took off when we introduced a new Service Management tool in 2007 including processes based on ITIL V2. It took a long time to implement, and initially it wasn't hugely successful. But what it did was replace a number of different tools and process implementations with a single tool and a single process for both the Netherlands and Belgium. For ING at the time this was massive: we never did anything together, and most departments took their own decisions for tooling. There was a Tooling and Methods department, but it wasn't hard to ignore them.

Then we got a new IT Governance structure. Processes were to be governed by no less than four layers: at the top the Service Management Council, with senior management from all domains involved (domain manager). Below it the Process councils with representatives at mid-level management (department manager) with a process expert. There were councils for Incident, Problem, Change and Configuration management. The third level was at domain level were the same mid-level manager chaired a meeting with all process managers from the departments within that domain. At the lowest level the process manager had his operational meetings with his process coordinators in the teams of that department.



Governance structure for Process Improvement

For Change Management on the operational level this meant departmental CAB meetings for every Service Management department, formally chaired by the Department Manager (or delegated to a team manager or the Change Manager), and a joined CAB meeting for all of DB/CIO chaired by the Head of Service Management, with representatives of every department. In this Change Control Board all high risk or high impact changes needed to be approved before deployment. Next to that there were separate process improvement meetings.



Governance of the Operational Change Management Process

An attempt was made to implement the same governance structure on the development side based on the CMMi framework, but due to the complex implementation that was chosen with a huge number of different processes (not unlike ITIL V3!) it never got the support or adoption rate as the Service management structure.

When we started preparing for the migration I was a Change manager in a department of three teams in a domain of three departments (Channels). By the time we were really up and running we had reorganised into five of the six (service management) departments of what was going to be IT of Domestic Bank Retail NL (the sixth joined us mid 2012). And by now I was the lead process manager representing the department in the process council, first as the expert, later as the sole representative of my department. One of the first tasks we undertook was to make proper process documentation in the form of Standard, Rules and Guidelines, signifying what was compulsory and what was good practice and how we wanted to work together. A huge part of that document was copied from the way my own team had been working previously. For the first time we had a single description of the process, the roles and responsibilities, instead of 10 different ones. The next step was to consolidate the non-functional requirements. First within my own division, later using that standard as a blueprint for the rest of ING.

At the same time senior management had asked the Change management community to think about ways to improve the collaboration between development and operations parties. They were also concerned with the number of waivers requested to deviate from the standards. Although this usually meant quicker delivery into production for our customers it also meant a potential increase in operations cost in the future. This proved to be the momentum we had been looking for to push our non-functional requirements out into the rest of the organisation as Generic Acceptance Criteria. In the design of this so called Tollgates process I collaborated with my CMMi colleagues to avoid checking the same thing twice. Once by the

project and once by the acceptant on the service management side. Looking back this was the first step towards Lean/DevOps: we eliminated double checks and instead had both our own requirements we were responsible for. And we were forced to trust each other's professionalism in delivering the goods!

Starting the journey

Lean and Agile principles get introduced into the game

In 2010 the first development teams started experimenting with Agile/Scrum. There had been a history of XP (eXtreme Programming) in that department to improve output and Agile/Scrum seemed the logical next step. The first steps were hard and didn't seem to deliver a lot. But after a few sprints velocity increased. And other teams saw how happy the people in the Scrum pilot teams were. So it started spreading. By 2012 Agile/Scrum was the official way of working for the development teams, and it became increasingly clear that management was aiming for a merger between the development and the service management teams. Some teams had already started with some Lean service management as early as 2009. The operations departments on the business side had had quite a bit of experience by that time with Lean, and some of the Blackbelts that had been improving the flow of business processes (like 'opening a new account') started analysing the flow within the service management departments. Senior management started talking a lot about Agile and Lean, and even the first copies of the Continuous Delivery book started showing up. All managers were sent to a Lean boot camp and were made responsible for implementing Lean Service Management in their own department. The service teams started using Kanban boards to track the progress of their incidents, problems, and changes.

Moving from process to content

Around the same time Peter, the manager of the Service Management department started pushing people to read Continuous Delivery by Humble and Farley. In our Change Control Board I teamed up with him to get people to start on deployment scripts, monitoring and automated testing. I had worked closely with him on the Generic Acceptance Criteria before and we knew we made a good team together. Peter as the visionary, pushing for change, me as the process man dotting the i's, making sure everybody followed through on their promises. At the same time he started challenging people to make their change approval decisions based more on the content of the changes rather than whether the process steps were followed. For a lot of my colleague Change Managers this was new, because they were trained to check mostly if people followed the process. Not everybody knew a lot about what their department was actually changing. The same went for the process managers for the other processes. A lot of the Incident managers were very good at managing the expiration dates of the incidents in their queue but wouldn't be able to solve an incident themselves.

In retrospect the next step was obvious, but for me and a lot of my process management colleagues it came as a bit of an unpleasant surprise anyway: by September 2011 it was announced we were getting rid of the process management positions. All employees should be able to work with the ITIL processes now and needed to take their responsibility for their actions. There was no more need for fulltime process managers. As it turned out, management was right. An analysis I did over the incident and change records of the period of transition from 2012 up to June 2014 showed no significant increase in the number of incidents. Of course the number of incidents on its own isn't proof enough, but the parallel increase in number of changes at least proves that we are more successful in doing changes. My work as lead Change manager must have made an impression with senior management, because I was promoted to Process owner. Now I really was responsible for a smooth running Change process within DB/CIO, without having the day

to day operational responsibility for Change management in a department. The move towards content over process in Service management as well was a logical step in light of the adoption of Agile/Scrum by development. For some time preceding the exit of the process managers senior management kept asking us whether a decision was based on content or on process; whenever the answer was 'process' they would emphasise the need to look at the content.

Introduction of DevOps

Still, these steps didn't solve our original problem completely: we still needed to improve quality, reduce cost and improve our time to market. The next step was that in Q3 2012 the announcement was made that from May 2013 we were making the transition from two organisations of Agile/Scrum development plus Lean Service management to a virtual DevOps organisation based on Agile/Scrum methodologies, leading up to the ideal situation of Continuous Delivery, using a host of state of the art (preferably Open Source) tools. Continuous Delivery was impossible in the current organisation because of the inability of the Service teams to cope with a faster pace. Development was Agile, but after a number of sprints the result was being handed over to a service team for integration and testing and delivery into Production. In effect we were doing a very classic 'scrum-fall'.

No more! The DevOps teams were going to be responsible for both new functionality and stability of production as a team; every sprint should lead to shippable (i.e. production-deployable) software. Whether it would be deployed would be up to the Product owner, as a representative of the Business. A team would consist of an average of 6 DevEngineers and 2-3 OpsEngineers. Because most value chains run across more than one team a coordinating role of Integrator was also created, together with a Blue Print Expert for more of an architectural viewpoint. Dev and Ops still had their own line management though, although in theory the teams would be self-organising (as per Scrum) under a Scrum Master. The role of Scrum Master was to be absorbed by the (Dev) Engineers in time.

All of the 1800 employees of DB/CIO had to apply for one of those positions, including team managers. And for all positions the emphasis was on technical skills. Only department managers and up remained were they were, along with me and my change managers for all six departments (plus a handful of other support staff). A current lack of technical skills would not be a barrier, as long as people had both the willingness and the ability to overcome the perceived gap in the next 18 months.

Naturally this announcement made some people nervous, especially in the service teams, where there was limited knowledge of Agile/Scrum, especially in some teams where the level of technical skills was low. People had a good functional (banking) understanding of their application (which you could argue should have been on the business side), and they had a good network to get problems solved. But they did not have in-depth knowledge of the platform it ran on, what the architecture was, what database it ran on, what communication protocols were used to communicate with other applications etc.. People asked to be part of one of the pilot teams that were set up to get some experience; other people tried to set up their own pilot teams. Internal Agile/Scrum training days were very popular, as were the introductory presentations on Continuous Delivery. When there was a seminar on DevOps in Amsterdam about 50% of the participants that day were from ING, including me, my team manager, her manager and a host of other colleagues.

Getting started

Getting back to the office after that seminar I thought about my search for knowledge and that of my colleagues. I decided that what we needed was a community on our internal social network that had just started, where we could talk about the transition and share documents and blog posts from all over the

internet. So together with two colleagues I opened our community and started posting links to interesting blogs, copying white papers and sharing relevant presentations from Meetups and DevOpsDays. The community has been running for over 2 years now and without any real communication about it we now have one of the largest active communities with close to 500 members from not just DB/CIO but all of the other departments in the Netherlands and most of the IT departments in other countries.

One of the people we met on that seminar was Kris Buytaert, a DevOps and Open Source evangelist from the earliest hour. We really liked his down-to-earth approach to DevOps, which stood out from the other presentations which were mostly by sales people. When we explained to him what we were planning to do and asked for his help as a consultant he thought we were joking, especially since the department that we wanted him to get started on was heavily into Mainframe. DevOps in a bank, on Mainframe and starting with 150 teams? We certainly picked a challenge! Still we insisted that he should come over to talk to some of our management and finally we convinced him that our motives were genuine and we really could do with some help. His help was perhaps subtle. He talked a lot to middle management, convincing them that it could be done. And helped a few talented engineers to set up some sort of rudimentary Continuous Delivery Pipeline, which could be presented in a DevOps training to their colleagues, creating momentum for the transition in the Mortgages department.

Joining the DevOps community

Through Kris we also discovered DevOpsDays. A couple of colleagues went with Kris to DevOpsDays Paris and came back overflowing with ideas. The first one was organising an internal DevOpsDay in June 2013. It had the same set up as the normal DevOpsDays, but all in one day, with some external speakers (which included some big names of DevOps who were in town for the DevOpsDays Amsterdam), some internal and in the afternoon Open Spaces.

Organising our first internal DevOps Day was a big thing!

The Open Spaces with great open discussions were a revelation to a lot of people. We liked it so much that after a couple of months we organised another one, but now with demos of tools instead of talks; a few months later we ran a 'DevOpsDiscovery' with small workshops where people could actually 'feel' some of the tools and get answers to some of their questions from colleagues who were already using them. In 2014 we had our third DevOpsDay, combined with the Amsterdam DevOpsDays, where I did a talk on our transition. Through the DevOpsDays organisers I and a few of my colleagues also became regular visitors of the DevOps Meetup group in Amsterdam, more recently contributors and sponsors.

Open Spaces were new to most of us, but we soon learned to love them

Ambitions for the future and problems we are facing

We have made tremendous progress! Every conference I visit I am reminded that we really are doing wonderful things! Within DB/CIO we now have 180 DevOps teams running Sprints of 2-3 weeks, resulting in Potentially Shippable Increments. Ideally we ship every Sprint. We have made an enormous step towards Continuous Delivery, with automated building, testing, deployments, with monitoring. We are making good progress on automated provisioning in our private cloud. Not all teams have made the same progress, but all teams have made a huge step. In all fairness, some teams had it a little easier than others, but there is room for improvement: not so much on the tooling side, but more on the organisation side.

Hearing all voices

Our transition was very much a Top Down approach. Senior management invested quite a lot of time to convince employees that the journey we were embarking upon was exciting and well worth the effort. However, not everyone is ready to move at the same pace, and many of staff had seen a lot of reorganisations over the past years. Why would this transition be any different to the preceding ones, that didn't really change anything about how we did our work but only replaced management? So change can be hard, and it takes time, more time than we sometimes realise, especially if you have been working on

the transition for so long and most of the people you talk to are very enthusiastic about it. Make sure you also hear the people who are not totally convinced, and keep asking for their feedback. They may think it harmful for their career to speak up, because they know that more reduction in staff is inevitable.

One message, made relevant a 100 ways

It is vital that all layers of management have the same message. The problem you often see - one that is common in large organisations - is that the message gets watered down as you pass it downwards in the hierarchy. It is impossible for a manager of 1800 people to talk to each and every one individually on a regular basis. It is even difficult to talk to all of his team managers on a regular basis; every time the message passes one layer of management it gets digested, interpreted and retold in a slightly different way if you are not careful. If a manager fully understands what we are aiming for and knows what that means for his department and is able to interpret the message in a meaningful way there is no problem passing it down; luckily most of our managers do understand. But we have to be realistic – just like some employees some managers may have trouble fully understanding why we are doing what we are doing and what it is we are aiming for. They are experienced managers so they still know how to manage a team. But they may not be able to inspire their team to reach their full Agile potential.

The balance between manager and coach

It is important to keep focussing on coaching your teams rather than managing them, which is tough when team leads are selected for their technical knowledge. They need that technical knowledge to be good coaches, but this doesn't always come with good management skills. Coaching does not mean getting up to your knees in the mud of content, solving your teams problems for them; instead, You should be coaching your team to solve their problems themselves, even when that sometimes takes longer than when you would be directive.

Changing our environment

In Agile/Scrum teams are meant to be self-organising, and so far the results seem to suggest that to be true: the most successful teams in the transition are those that can organise their own work with minimal interference by management. But at the same time we haven't changed our entire bank; some of the departments we work with don't understand Agile and we still launch 3 year programmes with very ambitious goals. How are we going to get those programmes embedded in the Product Backlogs of all our teams? And in such a way that they are not too dependent on the outcome of the other teams? How do we stop management from pushing for results on these programmes when we still make management responsible for delivery? We still need to do regular internal audits. Have we updated the rules we have to follow to our new way of working? And if there are audit findings, why do we make a manager responsible for solving it? Why not the Product Owner of the application? He is the one to put it with proper priority on the backlog!

Collaborative communities

Agile/Scrum teams are focused on delivering quality software that is actually working at a high but sustainable pace. That almost automatically means that retrospectives are internally facing: how can the team perform better, how can we work together more efficiently, etc. For a large organisation like ours that is not sufficient. Multiple DevOps teams have to work together in one value chain for our clients.

There are virtually no teams that can deliver new functionality to their stakeholders without touching other teams. Yet cooperation between teams is wanting at times.

The Scaled Agile Framework may help with delivering larger chunks of functionality (as an Agile replacement for traditional Programme Management). So may organising a Scrum of Scrums. We even have seen 'super' Product Owners being appointed to keep an eye on the bigger picture of combining different Product Backlogs in a Value Chain. But if this also works for solving incidents? And what about knowledge sharing? Yes, all teams are using Confluence to keep track of their team knowledge. Some engineers even know how to find some information on there from other teams, but a lot of teams operate as silos, inventing stuff themselves.

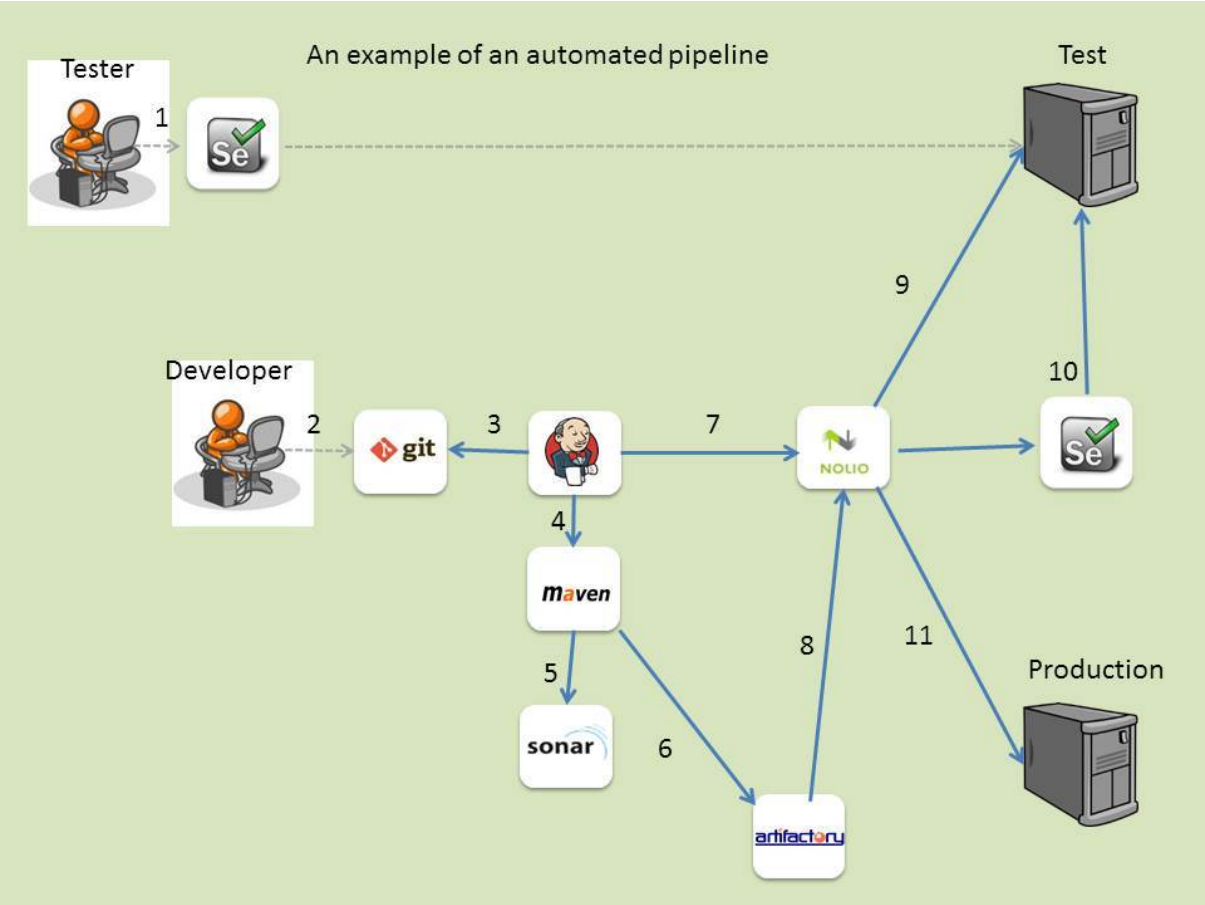
Through our internal community and through organising events like Meetups and Hackathons we encourage people to share, but I know we are not reaching everybody yet and maybe we never will. It will take a lot more effort and a lot more broadcasting before we can safely say that there really are collaborative communities of engineers, who share knowledge freely, even when it doesn't directly benefit their Sprint Backlog.

Sharing knowledge at an Open Space in our 2014 DevOps Day

Ambitions for the future

We are now almost 2 years into our journey towards Continuous Delivery. The other IT departments in the Netherlands are now also starting with the adoption process, including the infrastructure teams. For some of our own teams the road to Continuous Delivery has only just begun: they have only just begun using automated deployment and testing. Our ultimate goal is that all teams have a fully functional Continuous Delivery Pipeline, where code is built automatically upon check-in, after a validation that it lives up to the coding standards. The executables will be placed in an artifact repository with all necessary configuration files. The artifacts will be deployed to the test environment (which may be provisioned on the fly in our private cloud solution), tested automatically both for new functionality and for regression testing. After passing regression testing, the artifacts will be promoted to the Acceptance environment

where the acceptance testing is done. If acceptance testing is also passed successfully and there are no remaining issues with the Non Functional Requirements the new version will be released into production, if the switch for automatic updates is on 'yes'.



An example of a Continuous Delivery Pipeline

For the Non Functional Requirements we are currently building a tool where the teams will update the status for each requirements and be able to store all test evidence as well. This will help us maintain control over all our applications from a Risk Management perspective and aid in delivering the required in control statements to our regulators. At the same time it will automate most of the current Change Management process, facilitating the teams towards the goal of Continuous Delivery.

About the contributor

Jan-Joost Bouwman *is ITSM Process owner for Service Transition and Service Operations at ING in the Netherlands. He has significant experience in application and process management within large organisations. In his spare time he enjoys cooking and birdwatching (although rarely at the same time).*

Jan-Joost Bouwman

Trust, configuration management, and other white lies - Martin Jackson

Martin on Twitter: @actionjack[122] - Martin's blog: UncommonSense[123]

A UK Government Exemplar Service[124]

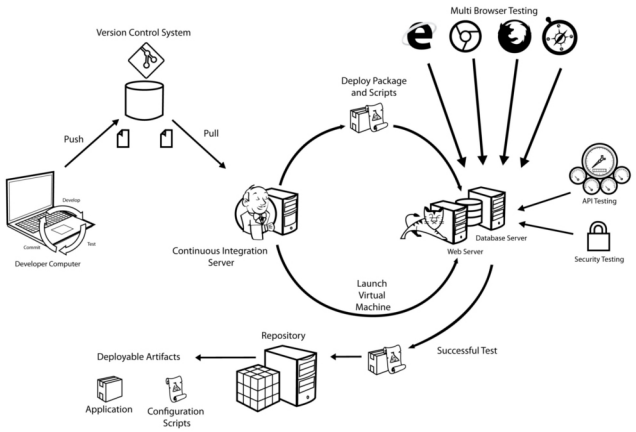
Timeline: October 2013 to present

The UK Government has 25 transformational 'exemplar' projects to create digital services that are simpler, clearer and faster to use. All of these meet the Digital By Default Service Standard[125] that was introduced in April 2014 and will be accessible to the public in March 2015.

In November 2014 7 exemplar services are live, with a further 15 in beta and the remainder in alpha.

How do you do Continuous Delivery and DevOps within the UK Government?

How do you do Continuous Delivery and DevOps on a UK government project with strict controls that prevent the majority of the team having direct access to the environments that they develop on, deploy to, and support around the clock?



Continuous Delivery

I am currently working for Equal Experts[126] on an exemplar service within the UK Government, and in my experience you can do Continuous Delivery and DevOps in such a setting by establishing trust within the team and with the client. Here are some of the things I have observed and experienced whilst working in this context.

[122] https://twitter.com/actionjack
[123] http://uncommonsense-uk.com/
[124] https://www.gov.uk/transformation
[125] https://www.gov.uk/service-manual/digital-by-default
[126] http://www.equalexperts.com

How can trust be weakened in CD and DevOps in general?

In a high velocity software delivery environment, trust between team members and between teams is vitally important. Unfortunately, trust can easily be worn down over time by the cumulative effect of many seemingly innocuous actions. Let's look at some of these trust-reducing problems.

Killing Me Softly

Every time someone logs into a box interactively I trust that box a little less. I don't know what was really done to the machine and the odds are that the person who logged on can't recall everything they did either even if they have a command history. A command history can't tell you what changes were made in a text file, for example.

One Step Forward (Two Steps Back)

I have found that there are two main reasons for unpredictable deployments:

- Environments composed of Snowflake Servers[127] due to a plan, environment drift, or sheer bad luck
- Methods or tools used that vary between environments

If deployments of a particular component or application are unpredictable, I will always caveat failed deployments and rationalise them away as technical debt because of tool and/or environment differences. In other words, I distrust the toolchain and begin to make excuses for failures.

Man On The Corner

My trust in a deployment drops when it fails the first time because of that odd little corner case that crops up every now and then. You remember it? That odd little corner case has actually occurred more than four or five times in the past few months, but you can never quite find the time to get it fixed.

Just This One Time

"We must do an ad hoc command, process, deployment without the overhead of the release process just this once because it's urgent". How many times have you heard that? Every time you allow a manual deployment, trust is weakened.

Need For Speed

Monitoring tools can sometimes be quite tricky to navigate, and log aggregation systems can be quite difficult to search when the pressure is on. So people sometimes (gasp!) feel the pressing need to log directly onto production boxes (see Killing Me Softly) because it is so much faster. This indicates a lack of trust in the available tools, and in what the monitoring tool is actually telling us.

[127] http://martinfowler.com/bliki/SnowflakeServer.html

I've Got The Power

"If I'm going to be doing on-call support then I must have root access to check configuration files, disk, CPU, I/O and perhaps the contents of the production database, in case I get called". For some reason people believe they will be extra-awake at 3am when that production incident happens, and they will need root/administrator access to all machines (see Killing Me Softly). This suggests they do not trust their existing access rights.

I'd Do Anything For Love (But I Won't Do That)

Abusing configuration management tools (Puppet, Chef, Ansible, Salt, etc.) leads to a loss of trust in the tools and in the people who write the code. Just because we can use our configuration management tool as a Swiss army knife does not mean we should. Some abuses of config management tools I have seen include:

- Monitor every file in the /etc, /var and /xyz directory
- Un-install package A through to Z
- Ensure what version of every package in a system is or is going to be installed
- Shell out to other systems and check their state and wait on them to be ready.

This kind of scope creep leads to uncertainty, unreproducibility, and lower trust.

Bang Bang (My Baby Shot Me Down)

Every now and then the "Treating your servers like cattle, not pets"[128] metaphor will be raised and driven to its logical conclusion i.e. using Immutable Servers[129] to guarantee server configurations are never touched. Sometimes this even goes as far as removing any services that would allow me to log in interactive to a server. Phoenix Servers[130] are a huge improvement over Snowflake Servers, and sometimes we need to trust services are there for a reason.

My Definition Of A Boombastic Jazz Style

I have heard so many definitions of what it means to be Done:

- "It's Dev Done"
- "It's Done Done"
- "It's Done if it's monitored in production"

A lack of agreement of what Done really means is one of the most effective destroyers of trust.

[128] https://twitter.com/damonedwards/status/325031895245135872
[129] http://martinfowler.com/bliki/ImmutableServer.html
[130] http://martinfowler.com/bliki/PhoenixServer.html

Switch

One of the most common complaints I hear from developers is around the creation and usage of Feature Toggles[131]. Some developers feel toggles are difficult to implement when components are too large or tightly coupled, and that it increases complexity in the codebase.

I think of this argument as "creating feature toggles are hard and I don't want to do them" or "that bit of functionality is so trivial it doesn't need a toggle", and it indicates to me developers do not trust such an ability is required. When Feature Toggles are used to Dark Launch features it can mean the difference between a successful deploy with a disabled failing feature, and a complete rollback because the failing feature could not be switched off in isolation. When you have to rollback a release it's that much harder to get the next release approved.

How have we established trust within the UK Government?

What have we done to fix some of these issues within government?

Building tools and canned commands

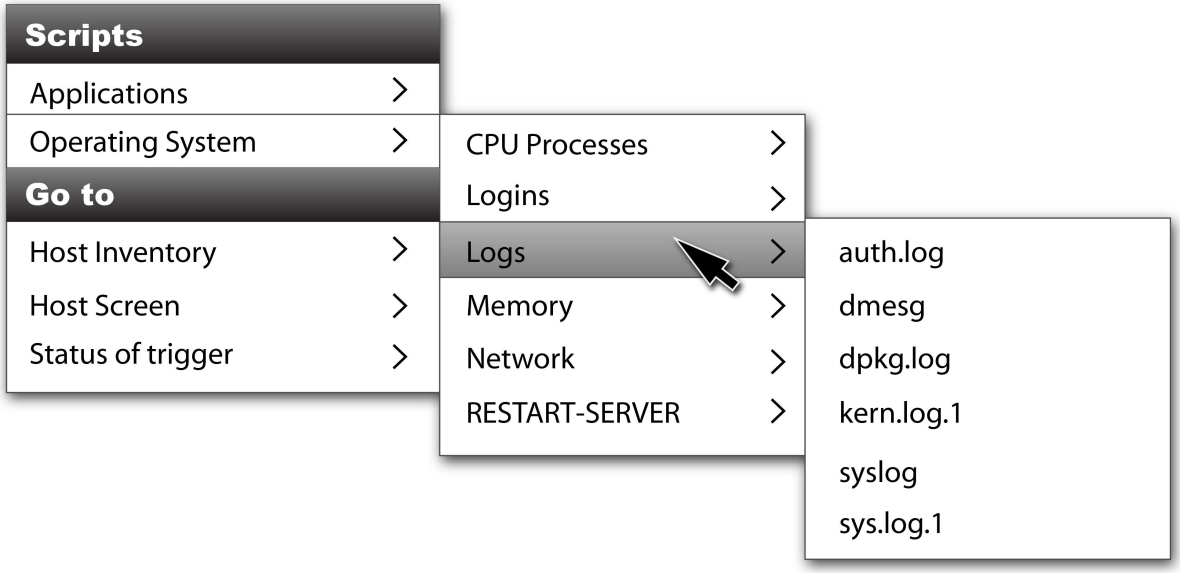
We created a interactive dashboard that hooked into a series of 'canned commands'. These canned commands or scripts only handle non-destructive operations so that anyone can monitor the environment and check configuration files securely on all services with limited interactivity.

As part of the implementation and ongoing maintenance we ensure that:

- Everyone has their own login and password, so we can restrict access and audit activity if needed.
- All service issues are reviewed, so we can identify any deficiencies in the canned command list and add them to our backlog.
- Custom programs are used to obfuscate customer-sensitive data e.g. usernames, passwords or API keys.

An alternative approach that uses canned commands with IRC is Github's implementation of ChatOps[132], which relies upon Hubot[133].

[131] http://martinfowler.com/bliki/FeatureToggle.html
[132] https://speakerdeck.com/jnewland/chatops-at-github
[133] https://hubot.github.com/



Example Canned Command

Ensure all environments are the same

One of the biggest problems we have faced is ensuring our environments are similar, let alone the same. We have had to address the following problems:

- Legacy implementations
- Lack of resources
- Cost
- IP address space limitations
- Singleton tools[134]
- Lack of multi environment support from third party services

We tackled this problem in several ways.

Forced environment refresh

Implementations (and mistakes) of the past came back to haunt us due to configuration drift, so by periodically refreshing and re-baselining our environments we made our configuration management process simpler (albeit by moving items from our configuration management code to our provisioning management code) by baking in new dependencies with more recent or even conflicting versions of software.

I was once told by a consultant that our current deployment was wrong because we do not recreate all servers from scratch[135] on every deploy and this was the only acceptable way to deploy an infrastructure that you can rely on. It was a arbitrary comment that lacked context, and ruled out any other methods for us to refresh our application let alone the impact, cost and overhead of adopting Immutable Servers. We questioned if we were doing the right thing by taking a hybrid approach, and our trust in ourselves was temporarily weakened.

[134] http://www.slideshare.net/matthewskelton/how-to-choose-tools-for-devops
[135] http://chadfowler.com/blog/2013/06/23/immutable-deployments/

Use of lightweight containers

Initially using virtualization we could rapidly prototype our development environments. Unfortunately, solutions solely based on hardware virtualization require resources and time, and as our environment grew recreating our infrastructure on our laptops became almost impossible - the cycle time to spin it up became unworkable and maintaining our custom laptop build became a painful overhead.

We have started to investigate lightweight virtualization container technologies (such as LXC, Docker, and Google Kubernetes) layered on top of hardware virtualization to create our complete environment on one virtual machine. This will require fewer resources, start up faster, and involve less custom configuration management code.

Create environments that can be switched off or recreated easily

The majority of cloud service providers charge by the hour and most development related activities do not occur 24 hours a day 7 days a week, so outside of those hours those resources are idle and you are burning cash that could be better spent elsewhere.

In order to contain costs and stretch resources, we decided it was worth the effort to invest time in generated short-lived environments for the following scenarios:

- Performance or load testing environments that are used once a day or week
- Debugging environments for finding production Heisenbugs[136]
- Experimental environments for testing hypotheses
- Temporarily extending our continuous integration capability on demand
- Testing disaster recovery procedures
- Validating that our configuration management code can do bare metal installations

By recreating our non-production environments regularly, we flush out any manual configurations or tweaks, making our testing more robust and reliable.

Proxy what you can but only when it makes sense

It has become a lot harder recently to get hold of fixed IPv4 addresses so initially we used reverse proxies to host multiple environments. However, we learned over time to limit the use of reverse proxies (or to make these reverse proxy implementations as close to production as possible) because of the risk of deploying configuration changes that did not accurately reflect production.

Whenever we deviated from an environment baseline we were always concerned since we could never really test or fully understand how specific changes related to our service and other third party services would be affected pre-deployment, and we considered this risk to be unacceptable. Eventually we took the hit and simply purchased more IP addresses.

Avoid costly common off-the-shelf software and hardware

One thing which saved us a lot of pain and suffering was sticking to our guns when it came to software and hardware stack selection. Experience has taught me to avoid any piece of software or hardware if:

- You can't put one in every environment

[136] http://en.wikipedia.org/wiki/Heisenbug

- You can't automate how you deploy and configure it
- You have to consider license costs before you can deploy it in a temporary development environment
- You have to constantly re-use, re-purpose and re-implement
- The last major version does not behave as expected
- The cost of changing your mind is outweighed by the cost of procurement

These criteria usually were mitigated by us favouring open source software in our designs.

Create stubs for third-parties that do not support multiple environments

Some of the services that we integrate with do not:

- Provide support for multiple environments
- Appreciate performance testing against their APIs
- Guarantee true API compatibility between their test and production environments
- Offer free API access

To combat these issues we stubbed out any services in our environments that did not meet all of the above criteria. Using these simulators allowed us to test and deliver services with a high degree of confidence, especially doing soak testing before allowing real customers near our systems.

Obsess over build and deployment inconsistencies

When a deployment fails in any environment other than development we force ourselves to obsess over the reason it failed and fix it. Deployment failures in later environments make everyone nervous and that nervousness translates in people lacking any real confidence in software successfully reaching production without issue. When confidence is lacking, getting the next release out the door is much harder.

Knowing it's OK to say no

We have a well defined, established and successful Continuous Delivery process that we use for all deployments, so why do people sometimes feel the need to try bypass it?

Well established Continuous Delivery pipelines can take time to complete and the pressure to get something out the door yesterday can be immense, so sometimes we have to ask ourselves is it worth running through the entire Continuous Delivery pipeline to do update something trivial e.g. update a single true/false option in a configuration file. However, is that change trivial or non-trivial? Trivial is one of those terms that tends to vary with who you ask but I boil it down to two things:

1. Is it possible to use an orchestration tool to push out the change to every environment while ensuring that the change is in our configuration management tool and has been validated for consistency (trivial),
2. Does the implementation and scope of the change using option 1 exceed the duration of a full deployment (non-trivial).

So, if someone requests a non-trivial change it must go through the entire release process. It is not really saying no - it's providing acceptable alternative options that you can have confidence in.

If your tools are getting in your way then get better tools

Inevitably one or more of your tools will bring you pain. This happened with us and chiefly for one or more of the following reasons:

- Our needs have outgrown the tool
- The tool is difficult to use
- The tool has been implemented in way which is suboptimal
- The tool itself has drastically changed
- The tool guru has left or entered the building
- There are newer and better tools about
- The tool does not scale as well as we hoped
- The cost of using the tool is greater than we were willing to bear
- The time investing keeping the tool running was greater than the time spent implementing something else

When one or more of these situations has happened we have either replaced or re-implemented the tool using knowledge learned from the previous implementation or other sources (e.g. the Internet, literature, and subject matter experts). We have taught ourselves to not fear change and also not to throw the baby out with the bath water.

Using a Safety Harness

Working in production scares me and it scares me the most at 3:07am when I am half asleep. In my honest option giving people the greatest responsibility and pressure when they are typically not at their best is perhaps not a winning combination. To combat this fear I use safety harnesses, which give me and others the power and ability to perform potentially dangerous commands with a degree of safety when we are both off- and on-call.

Our safety harness is an aide memoire that typically includes:

- A shared knowledge base of safe command sets
- Canned commands or scripts
- Rules of engagements i.e. Run Books or Work Instructions
- Limited access accounts for investigating issues

Production is a pretty unforgiving place sometimes and it less scary when you have a safety harness.

Expanding your tool kit

In the past we've sometimes been a little over zealous with our use of configuration management tools. We've pushed the tools beyond the boundaries of what they were originally meant to do which has caused us a few issues such as:

- Overly complicated deployment logic
- Excessive shelling out to Bash scripts
- Complicated service dependencies which may or may not be triggered e.g. database elections clobbering data migrations

- Handling what versions of what particular package was installed where but not with any great deal of granularity

These issues eventually led to hard-to-manage code with debugging dependencies when we tried to add new functionality, so we took a few steps back to make our lives easier in the following areas.

Configuration management

We implemented phased configuration runs (or stages) with specific configuration pipelines being placed one after another e.g. data migrations running before software updates. By doing this we sacrificed speed but gained greater reliability in our deployments.

We use remote execution tools to handle the ordering of deployment pipelines for services and service tiers.

Build scripts

We created environment independent build scripts for each application or service that worked on the Continuous Integration servers, locally on laptops, and/or in our throwaway virtual development environments. Doing this allowed us to be able to build our code locally and know it was being built by the CI server in the same manner.

Repository management

We now use a repository management tool to handle our upstream vendor dependencies to guarantee what version of what package gets installed where and when.

Everything in moderation

Some old hands in Operations may remember that the IT service mangement framework ITIL[137] had a non-prescriptive clause in it, which was completely lost when people rushed to implement every disciple without realising you could achieve 80 per cent of its benefits with 20% of the effort.

We've come full circle in the DevOps arena and have to learn the same thing again, since much of what we do are options and not prescriptions. for instance:

- We can compose infrastructures from a mix of Immutable Servers and Phoenix Servers
- Not everyone needs to deploy 10 times plus per day, we deploy once a week but our code base is always in a deployable state
- We don't all use the same processes i.e. our Developers use Scrum and our Web Operations team uses Kanban, but we use the same code base

Running in Production is the only true definition of done

I have heard many definitions of what it means to be Done but in my mind the only effective definition of done is to have the software successfully running in production and delivering business value. Any other definition of done is divisive since it separates the responsibility of creating, testing, and and delivering a feature, therefore making each stage in delivery someone else's problem.

[137] http://www.itil-officialsite.com/

Feature Toggles

On our team the Web Operations personnel deploy weekly using Kanban, while the developers use Scrum with two-week sprints. We are all continuously committing to the master branch while keeping the code releasable, and avoid exposing features in development using a combination of short-lived feature branches and Feature Toggles. New features are typically created on a feature branch behind a Feature Toggle, and then merged into master in an off state until ready for launch.

We have found Dark Launching features to be particularly useful when integration with a third-party provider is involved and testing beforehand is simply not possible due to provider constraints. For example, if a new feature relies upon a payment gateway that cannot be confidently tested in a pre-production environment then the ability to switch that feature off on production failure is very handy indeed.

Feature Toggles

Communication

Communication is key. However, it is important continually to emphasise that you can't have continuous delivery without environment stability and a known and trusted state, and we're aiming to guarantee safe outcomes (reliable deployments with can be depended upon). More importantly, it is not about not trusting your colleagues - we are in fact granting them greater access to more environments than they had before.

Conclusion

We have achieved a number of successes in our approach, namely:

- Trust from the business - our stakeholders are confident in the processes we use, since they are simple and transparent
- A team who feel confident going on call even with limited access production
- Skeptics and naysayers have become advocates - our approach which was initially perceived as unworkable has become the norm
- A business which was typically used to running large big bang releases is now used to small incremental weekly releases
- An increased appetite for change among the business community, deployment successes with incremental features has show that mistakes can be minimized and easily corrected

- To date we have high deployment success rate

A sign of success is that release parties are now a thing of the past - in fact, our releases are now pretty boring. To get here, we learned some valuable lessons:

- Communicate, communicate and keep communicating. You need to bring everyone along with you
- Be patient and unafraid to go over old ground
- Challenge assumptions and established processes respectfully. Try to understand why something is done a certain way
- Start small and show what is possible. Avoid extremely clever solutions and try to keep your solutions simple
- Try to use technology to empower those around you

In the near future we have plans to look at newer technologies such as lightweight containers and explore more automation opportunities. We want to build an automated system which can create, suspend, and delete complex multi-virtualisation environments for development teams. We plan to create infrastructure service tests to decrease the cost of operational acceptance testing, and migrate to an OS native package management system to make our deployment process and configuration code easier to manage.

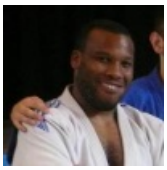
Finally, we have to translate and transfer the benefits of our approach to other teams and organisations within the UK government!

Song title references

- *Killing me Softly* by Roberta Flack
- *One Step Forward (Two Steps Back)* by Johnny Winter
- *Man On The Corner* by Genesis
- *Just This One Time* by Cher
- *Need for Speed* by Petey Pablo
- *I've got the Power* by Snap
- *I'd Do Anything For Love (But I Won't Do That)* by Meatloaf
- *Bang Bang (my baby shot me down)* by Nancy Sinatra
- *My Definition Of A Boombastic Jazz Style* by Dream Warriors
- *Switch* by Will Smith

About the contributor

Martin Jackson *is a DevOps Engineer and consultant for Equal Experts, DevOps advocate, Infrastructure as Code Hacker, and keen Judoka. His specialities include making the impossible possible, herding cats for fun and profit, and throwing people over his shoulder.*



Martin Jackson

Avoiding the pendulum swing - Rachel Laycock

Rachel on Twitter: @rachellaycock[138] - Rachel's blog: rachellaycock.com[139]

ThoughtWorks[140] - various clients

Timeline: July 2010 to present (November 2014)

ThoughtWorks is a global company of passionate technologists. They specialize in disruptive thinking, cutting-edge technology and have a hard-line focus on delivery. Their clients are ambitious, with missions to change their industry, their government, and their society. They provide software design and delivery, pioneering tools and consulting to help them succeed.

Context

A few years ago, a large retail banking client asked ThoughtWorks to help them implement Continuous Delivery (CD). They were on six monthly release cycles that were painful and fraught with risk. Release days required the entire development, testing and operations teams to come in for a whole weekend. Sometimes several if the first attempt wasn't successful. After reading the Continuous Delivery book by Jez Humble and Dave Farley an executive decided they wanted and needed to deliver more rapidly, to continue to compete in the market. We of course said "sure". We had been delivering software using the principles and practices of Continuous Delivery for years.

But…

Three months later we weren't getting anywhere fast and the client wasn't too happy. So what went wrong or what was holding us back? Their architecture, or what felt like a complete lack of architecture. Their code based was huge and complex - over 70 million lines of code and millions of dependencies, many of which were cyclic. They had the most common software design pattern that exists "the big ball of mud[141]" (BBOM). I realised that CD isn't just about build and deployment pipelines with lots of tests to give you the confidence to release all the time. In fact it relies heavily on your architecture. All the previous times I had been "doing CD" I'd been working with fairly clean architectures or on a greenfield project. Even better, the application was designed with CD in mind. So breaking our architecture into independently deployable pieces was a first class concern at the start. But what if you aren't so lucky, what if like my client you would like to "do CD", but you have a big ball of mud, what then?

Conway's Law

As I delved into how you could design your way out of this problem with patterns like the strangler or branch by abstraction or many of the patterns Michael Feathers described in his book on *Working*

[138] https://twitter.com/rachellaycock
[139] http://www.rachellaycock.com/
[140] http://www.thoughtworks.com/
[141] http://www.laputan.org/mud/

Effectively with Legacy Code[142], it occurred to me that even if you implemented patterns and made your design and architecture better now, what would stop it ending up in a different but ultimately similar BBOM? This is when I came upon Conway's Law[143], which states: "organisations which design systems … are constrained to designs which are copies of the communication structures of these organisations".

Wikipedia states: "Although sometimes construed as humorous, Conway's law was intended as a valid sociological observation. It is based on the reasoning that in order for two separate software modules to interface correctly, the designers and implementers of each module must communicate with each other. Therefore, the interface structure of a software system will reflect the social structure of the organization(s) that produced it." But how does this play out in real life? Let's go back to my retail banking client.

No more dependencies!

So they had the BBOM architecture, but they hadn't designed it that way. The BBOM pattern itself is usually "caused by expediency over design", but I met the enterprise architects and some years before they had in fact designed the system. In fact rewind 5 years and they had another application that served the same needs. It worked as their retail banking application inside of their bank branches. They had built up several components, which had to talk to each other to create the whole system. Unfortunately for them, they were on the Windows platform and this was a point when dependency management didn't exist. So as their system grew so did their "DLL hell[144]". Having been burned by this experience and the problems it caused both at release time and then in production, they decided to redesign the system from the ground up. This time they would avoid DLL hell by creating vertical slices through the system, from front-end to CRUD at the database level, for every piece of functionality. It started out well, the developers were moving quickly and they were releasing regularly. In the minds of the architects and to those around them it looked like a fine design job all round.

5 years passed later

Fast-forward five years, and significant dependencies within the system existed. As online banking became popular, the 'in branch' application turned into a web application. It then also morphed into an online business banking application. Now we have three apps with similar functionality. Do you think a vertical slice through the system was created for every piece of functionality? No: project managers would push back on recreating the same functionality. And developers - who had no insight into the reason why they created these slices - began to create dependencies in the system. Because these dependencies were not managed or the reuse was 'unplanned', as one architect stated, they became cyclic. Try getting that on a build pipeline.

So the architects designed the system, but they did not design the organisation, or reflect on the design as the requirements changed. In the end, the design reflected the problems the organisation had: a lack of communication structures for dependent pieces of the system.

Modularise all the things!

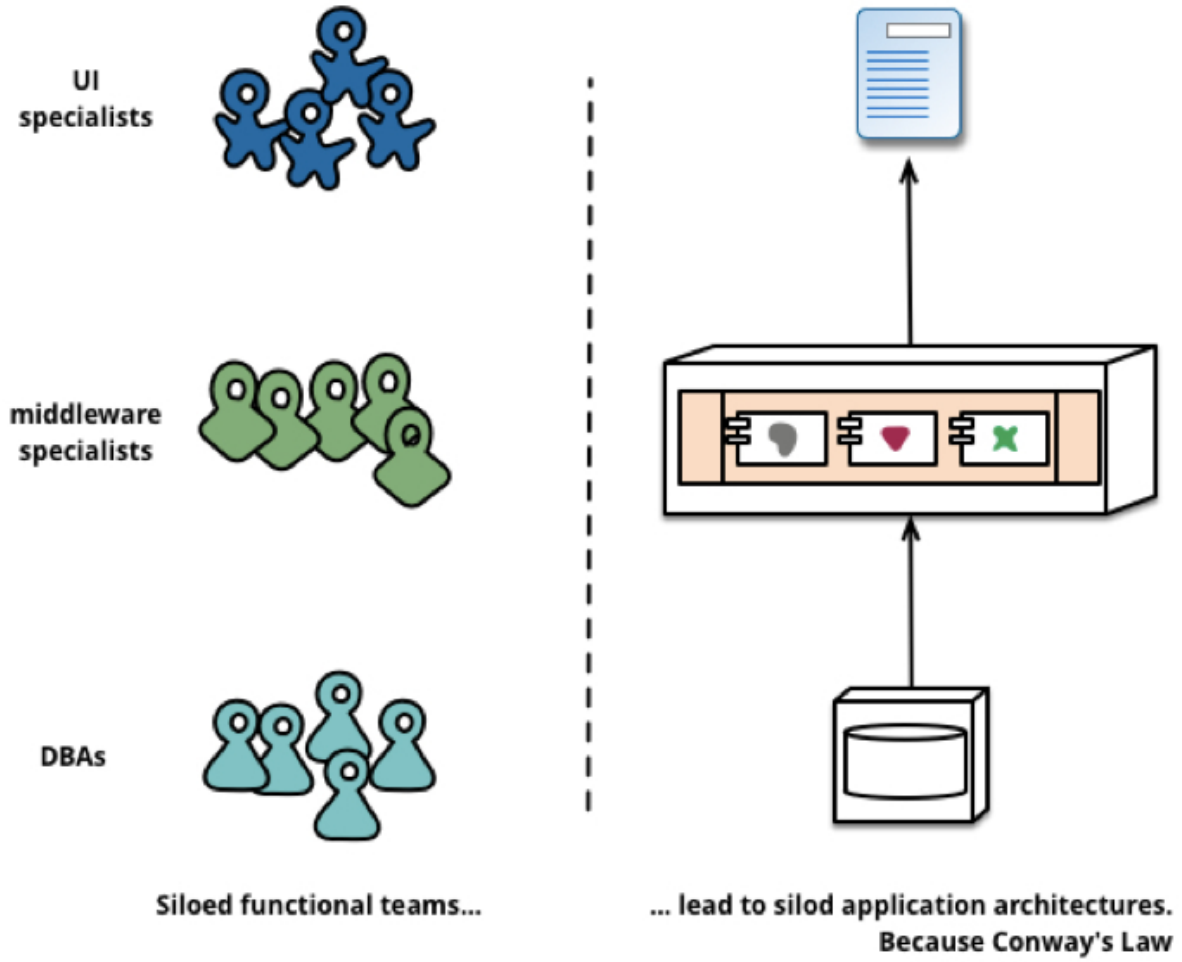
So you may be thinking, as many agile teams do, "sack the architects!". Many of the issues this organisation had stemmed from Enterprise level architects making decisions, that in theory would help

[142] http://c2.com/cgi/wiki?WorkingEffectivelyWithLegacyCode
[143] http://www.melconway.com/Home/Conways_Law.html
[144] http://msdn.microsoft.com/en-us/library/ms973843.aspx

the organisation deliver software more cheaply by standardising on the technical models. We often have an allergic reaction to this because it can stifle innovation. But what if teams could go ahead and make the right decisions for their teams and organise around their business capabilities. What if you break the system up into parts manageable for a team. That will solve your problem, right?

Death by autonomy

Another year, another client. They had issues with their monolithic software and were unable to get anything out of the door. This was due to the huge coordination effort of getting all the layers of the system running and communicating together. They decided to break it up so that each team was responsible for just one module or component, e.g. the CMS, the APIs or one of the myriad of front ends. The teams acted like a startup, the developers choosing any tool they wanted. They also wanted Continuous Delivery. This time, would you believe it, all the autonomy was slowing them down. Why?



Monolith

All the pieces (modules) still had to fit together and talk to each other. They also had the problem of QA being a separate organisation, an afterthought. At release time they would all pile into a shared environment for 'integration' testing, only to discover all the many ways that things were broken. The autonomy had caused such huge divergence in technical stacks that infrastructure's ability to support everything was causing a huge bottleneck.

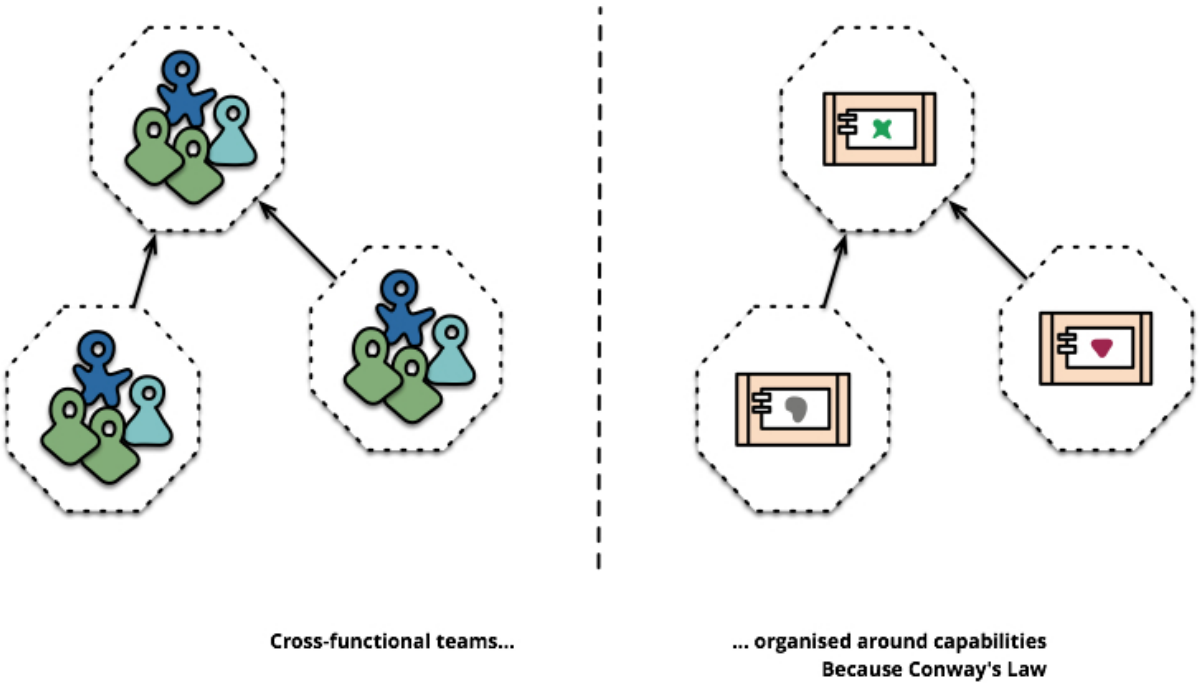
For the first time in my career I recommended that they look at their architecture as an enterprise. They had architects, but only application architects who were making siloed decisions. Many components were built again and again for different platforms, causing crazy levels of divergence in logic, and making the payment model for their system incredibly hard to manage. Because at some point they still have to integrate. This was another client who had swung from one extreme to the other in their design: once their system got bigger they had problems releasing all over again.

Stop the pendulum swing!

Is this all sounding familiar? How do you avoid this pendulum swing of architecture?

It is time to get real about Conway's Law: it isn't some humorous anecdote. Time and again I see this play out from client to client, team to team. There is no one true design or silver bullet for your architecture; you will need to break up your design into components or services. Most designs become too big to fit into any one teams' heads, but the trick is to *split things by intent*. Design your organization and the teams within it to match the architecture you want, otherwise you'll just end up with an architecture that looks like your organisation anyway.

You should definitely create multi skilled teams to the best of your ability, because these prove to be the most effective at solving problems and moving quickly. But those teams will integrate with others in the enterprise. Wherever integration points exist, you have to work extra hard on creating good communication structures in your organisation, especially if you want the interfaces and communication methods to make sense and be simple in your design. Creating affinity groups by specialisation outside of the teams helps, as do 'scrum of scrum' types of activities. Structure your teams how you want your software to look and create communication structures for the interconnected pieces.



Conway's law in action

You have to take into account continually the constraints, the value of your product, how often you want to release and what you can afford operationally. Many things we know and have known about software design and architecture stand true.

Parting thoughts

I cannot criticize any of my clients, not just because they pay the bills, but because we all make decisions like these. Teams that designed the systems I described were full of smart people with many years of experience, so why do they keep swinging from one problem to the next? Because they are still trying to solve the wrong problem. The problem is that there isn't one solution and you are never done, you are always re-designing. So given Conway's Law the only real way to 'design' any system is to design as flexible an organisation as you can.

And **allow your organisation to keep designing your system**.

"Because the design that occurs first is almost never the best possible, the prevailing system concept may need to change. Therefore, flexibility of organisation is important to effective design." Melvin Conway

About the contributor

Rachel Laycock *is a Market Technical Principal for ThoughtWorks New York. Her main role is that of a Technical Lead and Architect, but since joining ThoughtWorks she has also played the role of a coach, a trainer and a project manager. Alongside this she is the Editor-in-Chief of the P2 Magazine*[a] *Rachel is also a member of the ThoughtWorks Technical Advisory Board, which meets regularly to produce the ThoughtWorks Technology Radar.*

Rachel Laycock

[a] http://thoughtworks.github.io/p2

Communicate to collaborate - Matthew Skelton

Matthew on Twitter: @matthewpskelton[145] - Matthew's blog: matthewskelton.net[146]

A major UK-based ticket retailing website

Timeline: 2011 to 2013

Context

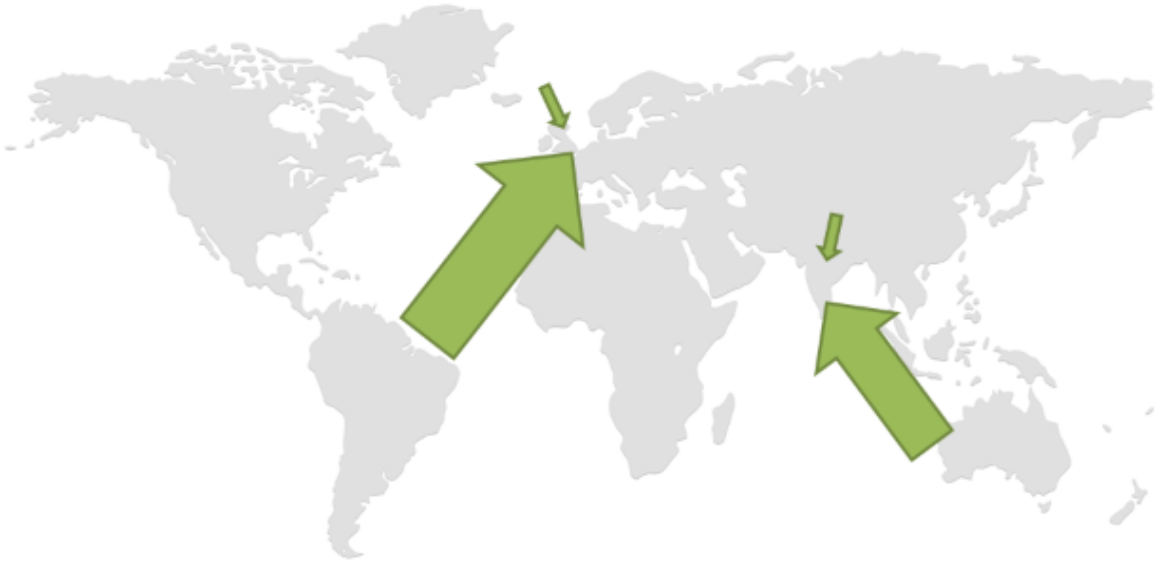
The organisation began selling tickets online in 1999, making the website one of the longest-running online booking websites in the UK.

From 2010, the organisation embarked upon a major transformation of its technical capabilities across all areas of IT: development, testing, release engineering, system operations, and infrastructure. The company grew the UK-based teams to 5 development (Dev) teams, 10 application support people, 12-15 people in infrastructure and operations (Ops), and 6 in IT Support. In addition 200 developers, QAs, environments admins, etc. were retained in Bangalore through ThoughtWorks and were part of the internal team. By the middle of 2012 the teams looked like:

- 10 x 2-pizza dev teams - London and Bangalore
- 1 x 8-person NFT (Performance Testing) team - Bangalore
- 1 x 10-person Regression Test team London
- 2 x 2-person Build & Deployment teams in London and Bangalore
- 1 x 8-person Environments team - London and Bangalore
- 1 x 5-person Internal IT infrastructure team - London and Edinburgh
- 2 x 8-person front-line system support teams - London and Bangalore
- 1 x 8-person IT Operations - London

At least 260 technical (hands-on) people were involved in building and operating the software systems at this organisation, and I joined the company in late 2011 as Build and Deployment Architect in order to lead the build & deployment activities as part of this transformation.

[145] https://twitter.com/matthewpskelton
[146] http://blog.matthewskelton.net/



10+ Dev teams in two locations (India and UK)

This article focuses on the ways in which we within the Build & Deployment team in London nurtured communication with other teams in order to improve collaboration and improve the way in which the software systems were built and operated. We were inspired by the 2010 book *Continuous Delivery* by Jez Humble and Dave Farley, and by the emerging patterns from the DevOps movement, and sought to use these and other practices to make software delivery more reliable and predictable.

Problems and causes: mental and financial models, execution, and communication

By 2011, the commercial teams were increasingly concerned that the cost and speed of software delivery were too high, and they began demanding improvements from IT. Predictability of feature releases was paramount for the commercial teams, who had relationships with commercial ticket resellers, large corporate (B2B) clients, and other business customers. Features would be promised contractually to clients for a specific date, so reliability of delivery was more important than pure speed in a market dominated by a few companies.

The working assumption was that it was possible to specify features independently of other features without negative future implications. Over time, this approach had led to a system with significant technical debt that made development of new features ever more costly, a large central monolithic database, and a low level of focus on operational concerns such as logging, metrics, diagnostics, and deployability. This created a lengthy regression testing phase and infrequent big bang releases (a platform release) every three months or more. As platform releases became larger and less frequent they became increasingly error-prone, leading to a drive for even less frequent releases despite the commercial teams pushing for more and more features[147].

A 'smell' that indicated something was amiss was the presence during 2010 of a small development team that was empowered to bypass the lengthy platform release process and work on innovations in their own workstream. Useful features outside the main platform were developed and released rapidly in close

[147] these patterns are common at many organisations, and certainly not unique to this organisation.

collaboration with the commercial teams. The development speed and collaborative nature of this small team were a welcome eye-opener for many, but unfortunately there was limited focus on Continuous Integration, automated deployments, or logging. After I joined it soon became clear that within our Build & Deployment team we would need to spend significant time increasing awareness of good software practices within the wider organisation if we were to be effective.

Treat the build system as production

When I joined it's fair to say that the state of Build & Deployment was not very healthy. The quality and reliability of builds and deployments had become gradually worse: builds would break for no obvious reason; deployments would fail apparently randomly; and teams were contending for shared environments in which to test their changes. Pressure to get things working had led to developers with administrative access to test environments, resulting in tests going green due to undocumented fixes. My colleague had been given the mostly thankless task of getting the builds green, and likened these activities to juggling flaming plates.

The Bangalore and London teams each had their own set of build and test infrastructure[148]. The London system initially included ~130 servers:

- 4 servers running Go[149] for CI and deployments
- 100 build agents, some doubling as test agents (!)
- 5 machines for data replication
- 4 Subversion servers for source code and artifacts
- ~20 other machines for testing

This infrastructure had grown organically over several years, and had no real owner except the (overworked) Build & Deployment team.

Be a Service Desk

I realised quickly that the build/deployment/replication (BDR) infrastructure was an internal live system, and that we needed to treat it as a live system to stabilise and run it effectively. Specifically, there were certain points in the delivery cycle when BDR unavailability would have blocked deployments to Production. On one occasion, a crucial deployment server in the primary data centre was 2 hours away from being decommissioned because it was not classified as Production. We recommended the machine be retained and rebuilt with Chef to prove we could survive a future decommissioning incident!

We setup a JIRA ticketing system to give us a single point of contact for issues affecting the BDR system. When people walked up to us, sent a Skype message, or emailed over a problem, we asked them (nicely!) to report the problem via a JIRA ticket, explaining that we needed to triage and prioritise the issues.

Crucially, we did *not* attach SLAs or response time guarantees to the ticketing system and only used it to centralise reporting. Submitting a ticket was as simple as emailing a specific address, and I also printed some cards with details of our JIRA ticketing system:

[148] for details of the infrastructure and activities in Bangalore, see the contribution in this book by Sriram Narayanan, my colleague and counterpart in Bangalore at the time.

[149] http://www.go.cd/

Biz card for build system problems!

The cards and the JIRA system seemed to confirm for people that we were serious about addressing the instabilities in the BDR infrastructure[150], and we began to gain the trust of development teams. We found using a light-touch ITSM approach based on aspects of ITIL®[151] also helped us to win friends in IT Operations because we were now speaking their language.

Collaboration with IT Support

A practical way in which we improved BDR was to develop a close working relationship with IT Support. We built up trust by helping each other out; we shielded the IT Support team from the more random requests from developers wanting a new tool, and in return they helped us to operate the BDR infrastructure by adding monitoring or configuring backups.

In order to emphasise the need for the BDR infrastructure to be taken more seriously, we collaborated on defining the service levels (recovery point objectives, backups, etc.) for all the key services and servers in the BDR estate:

[150] the team name Eccentric Ninjas was the first name that was suggested by an online team name generator, and it stuck!
[151] ITIL® is a Registered Trade Mark of AXELOS Limited

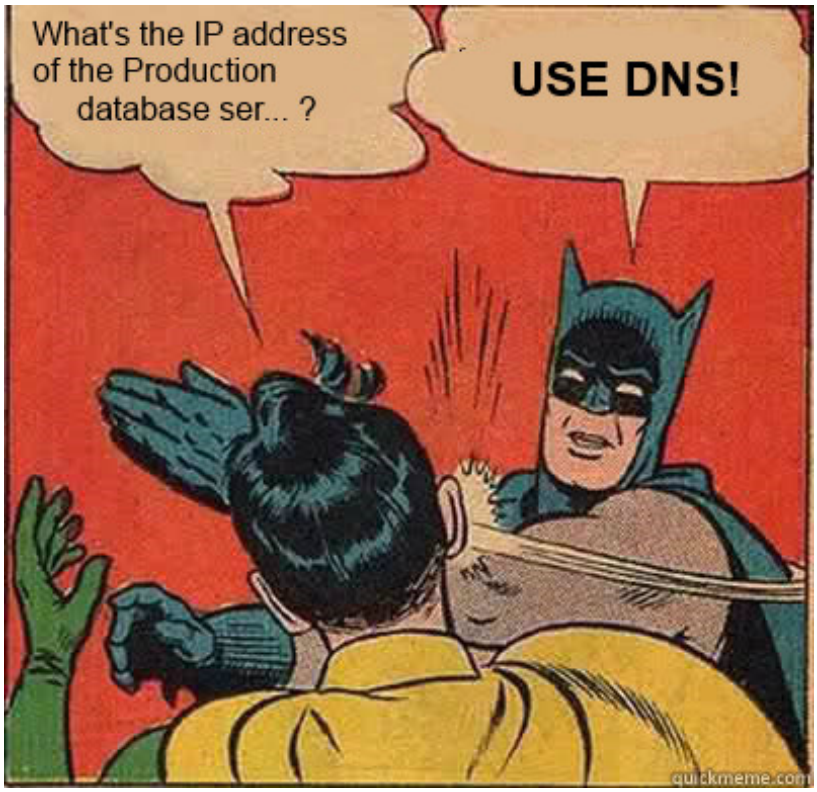
	Reference Data: (expand/collapse)						
	Add new items at the *bottom* of the list			Normal service: London office hours		Special service: specific times of the delivery cycle	
	Unique ID	Description	-	Normal Service Tier	Comment on Normal Service	Special Service Tier	Comment on Special Service
	[redacted]	Code		Gold			
	[redacted]	Legacy Code		Bronze			
	[redacted]	Fisheye		Iron			
	[redacted]	Replication GO		Silver		Gold	
	[redacted]	Replication server		Bronze	For much of the delivery cycle, this server is notused	Gold	During Pre-Prod cycles and Prod deployments, this server must be available
	[redacted]	Published Artifacts Replication		Silver		Gold	
	[redacted]	Bangalore - London MPLS Link		Gold			
	[redacted]	Log Search		Bronze			
	[redacted]	Build GO		Gold			

Differentiating service levels at different times of the delivery cycle

Shortly after this, we worked together on a highly-available monitoring system built on Zabbix, with API access for the Build & Deployment team to configure alerts and thresholds. Without the earlier collaboration on ticketing and service level requirements, the shared monitoring would have been much harder to achieve.

Use the build systems as a test-bed

A happy side-effect of having a large and complicated Build & Deployment system was that we could prototype new approaches within our own live system before suggesting new approaches for the main Production systems. For instance, in early 2012 the Production systems did not really use DNS for endpoint resolution and most people used specific server IP addresses. We knew this was hampering good deployment and service discovery practices, so we trialled a couple of approaches to DNS management within the BDR system before recommending an approach for other environments: using an environment-agnostic DNS entry in config files with the fully-qualified part being supplied by the Primary DNS Suffix for an environment.

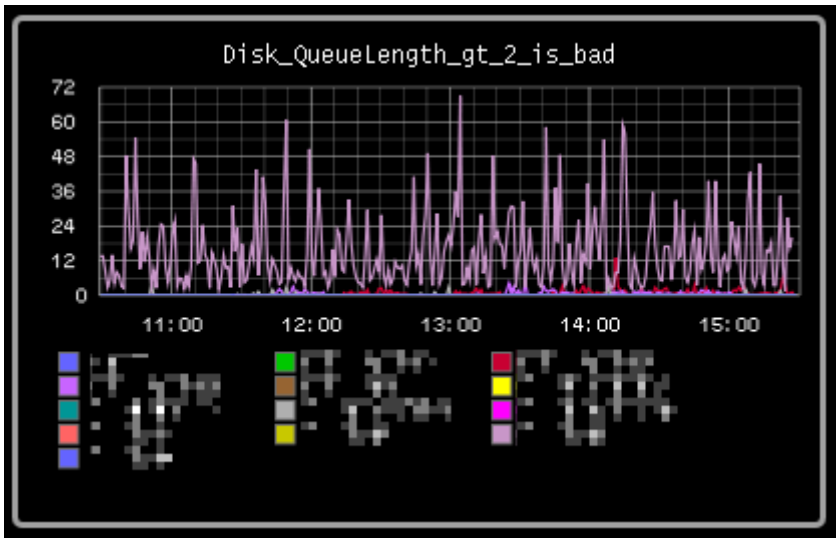


Use DNS, Robin!

In a similar way, we also trialled and demonstrated several other new approaches and technologies:

- Chef for infrastructure configuration
- Artifactory for NuGet, Chocolatey, and RPM packages
- Graphite for time-series metrics and monitoring
- The vSphere API for virtualisation
- Vagrant for manual and automated VM image changes
- Patching schemes for Windows and Linux
- Building and testing VM templates within CI

The introduction of Graphite (promoted by my colleague in London) was a particular success, as the existing monitoring tools used by IT Operations were end-of-life. One morning at 8am, a sharp-eyed developer spotted the metrics had hit zero and told the IT Operations team that something was wrong. Operations had not yet seen the problem because their tools aggregated data every 5 minutes, so they wanted to know how the hell we had found the problem first!



A mini Graphite graph from an unhealthy server

Improving the software release process

Use Conway's Law to align teams and software systems

One of the key changes we were deeply involved in was the move from a shared codebase model where any Dev team could work on any part of the system to a model where teams were aligned to sets of products & services. We called this product-aligned teams and did it for several reasons:

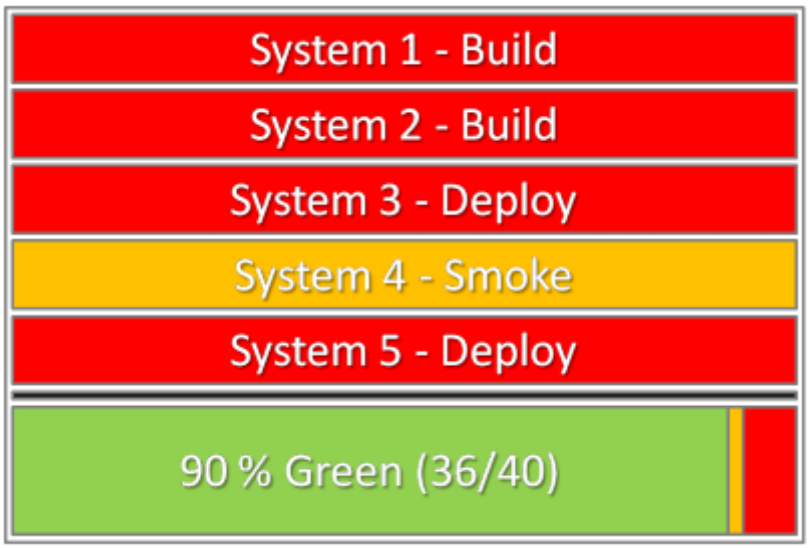
- We knew that we needed different parts of the system to be independently deployable
- Conway's Law[152] strongly suggested that independent subsystems would require teams to have responsibility for specific subsystems
- We wanted a greater sense of ownership over the code in general

We found volunteers (mostly willing) within each Dev team to act as go-to people for any build or deployment problems relating to a subsystem. It was their responsibility to coordinate the fixing activities with their team, whilst providing a point of contact for other teams with dependencies.

We hit an interesting snag when moving to product-aligned teams: when all broken builds were displayed on a shared information radiator, teams often ignored red statuses, even for their own components. We tried various different ways of displaying the information until we realised that because the radiator never showed any green at all, the *meaning* of red had become diluted: an interesting psychological gotcha. A developer consequently tweaked the build radiator software to show a percentage bar.

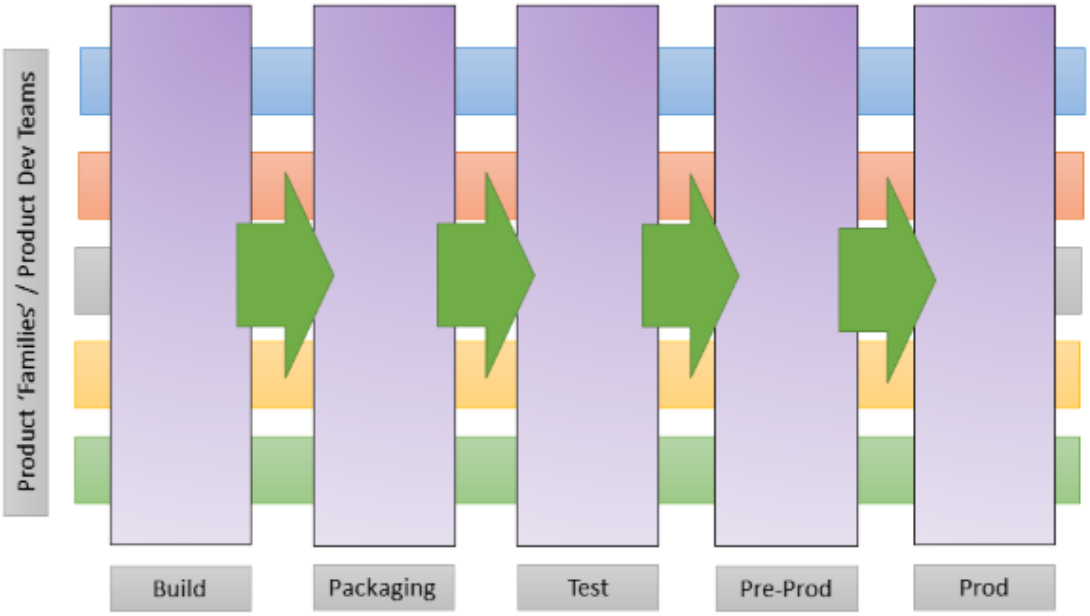
[152] "Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations". See the contribution by Rachel Laycock for more information on Conway's Law.

All red dilutes the meaning



Green percentage gives context

Applying Conway's Law to the build & deployment systems themselves, we began using a separate build & deployment server per team, forcing a separation of tooling between teams, in addition to a separation of application logic. Prior to the change, we had one deployment server for each set of activities (build, packaging, test, Pre-Production, Production), shown here in the vertical purple blocks:

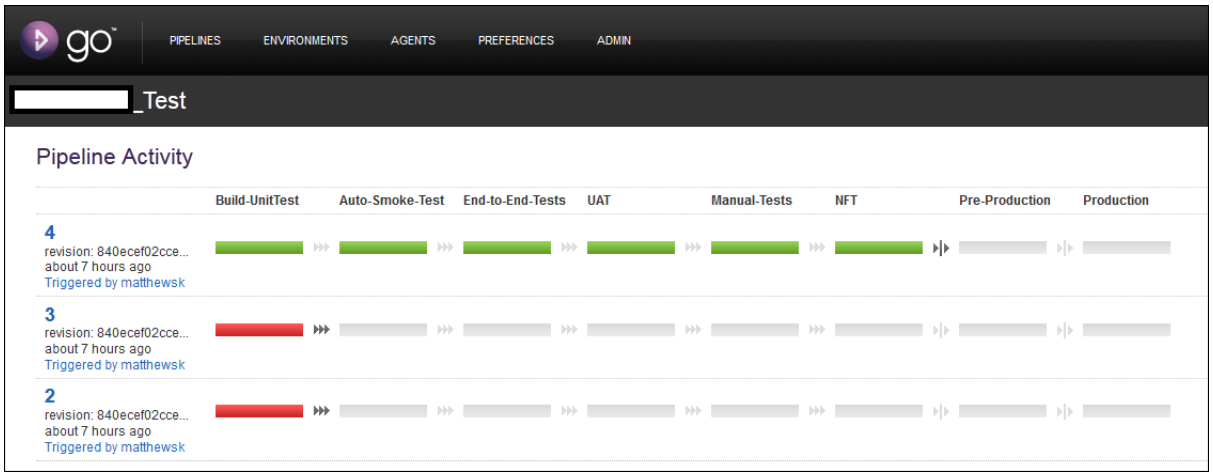


Deployment servers optimised for function

We changed the focus of the deployment servers so that each team had its own server (shown here in purple) that could take that team's software all the way from CI through testing to Production:

Deployment servers optimised for value flow

With the old approach, it was very difficult to trace a change from version control to Production and vice versa. In the new scheme, we could build a deployment pipeline that was simple and clear enough to show to non-technical people and have them immediately grasp the progress of a change:



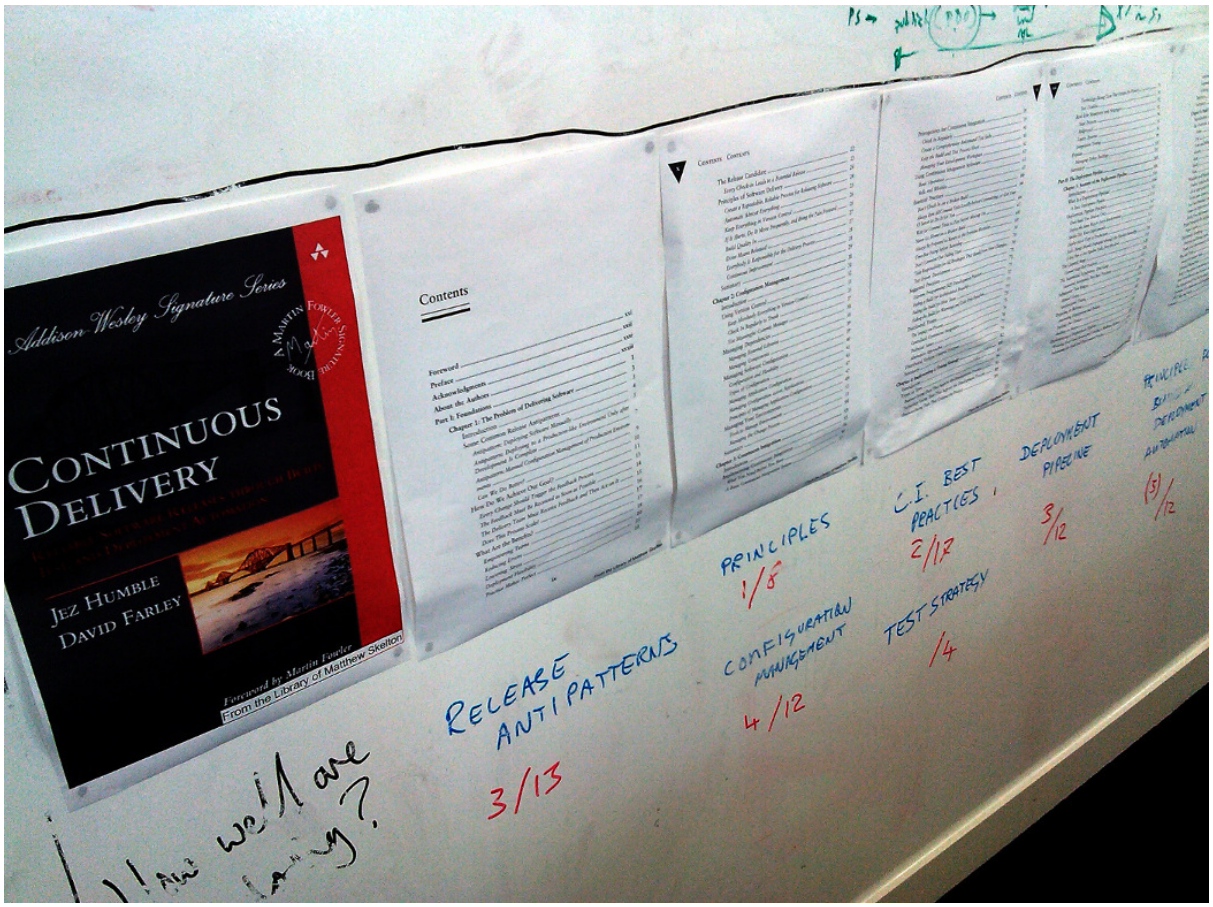
Deployment pipeline stretching to Production using ThoughtWorks Go

Improve quality incrementally

Having read (and re-read) the Humble & Farley book 'Continuous Delivery' I realised that the contents pages of the book read rather like a checklist for sensible technical practices. For example, some of the subheadings are:

- Every Change Should Trigger the Feedback Process
- Keep Everything in Version Control
- Never Go Home on a Broken Build
- Only Build Your Binaries Once

I printed out the Contents section on large paper, stuck the pages on a wall by a thoroughfare, and annotated each page with a score reflecting how well we thought we were doing. Quite a few people stopped by the printouts and asked about the rankings and the reasons behind the Humble & Farley recommendations, which sparked some useful discussions. This was a practical way to engage people in thinking about the scale and nature of changes we needed to undertake.



The Contents pages of 'Continuous Delivery' form a handy roadmap

A further way in which we worked with teams to increase engagement with the different software products and subsystems was to run a series of workshops with Dev and QA teams to characterise the kinds of testing activities they thought were necessary. By this point, members of the QA team had begun to work more closely with Dev teams rather than just as a QA team.

Essentially, we characterised our ideal value stream and looked at how the current state of the deployment pipelines matched that ideal. For each kind of testing activity we discussed a range of factors related to each test type:

- Duration (magnitude)
- Magnitude (num tests)
- Value Gained
- Scope
- Cost of creation
- Cost of maintenance
- Cost per test
- Visibility of test results
- What does stage failure mean?
- Number of machines
- Manual or Automated
- Duration (seconds)
- Technology or Business Facing?

- Purpose - Support Programming or Critique Product?

Exact values were not important; what mattered was that teams consciously thought about these questions in the context of the products and subsystems they were now responsible for. Here is an early version of some subsystem target deployment pipeline stages:

Stage properties	Build	Unit Test	Isolated Tests	Exploratory
Pipeline Meta-Stage	Commit Testing	Commit Testing	Auto Acceptance Testing	UAT
Duration (magnitude)	10 sec	1 min	3 min	1 hour
Magnitude (num tests)	10	100	30	5
Value Gained				
Scope	Solution File	In-memory, does not make HTTP call or contact DB	File System + Local DB	
Number of machines	1	1	1	3
Manual or Automated	Automated	Automated	Automated	Manual
Duration (seconds)	10	60	180	3600

We can see the teams felt unit tests should take no longer than about 60 seconds, whereas integration tests might take 3 minutes, and exploratory tests might take an hour to complete. We also captured the principle that unit tests should not make an HTTP call or touch a database, and discussed other kinds of constraints. We were particularly influenced by the outside-in 'GOOS' model of system testing[153]. The members of each team also rated the test stages for their own subsystems as Green (healthy), Amber (needs work), Red (problematic), or Grey (not relevant), based on gut feeling and analysis of Go logs:

	A	AA	AB	AC	AD	AE	AF	
1	Component / Subsystem / Product	Current Pipeline Stages						
19	Name	Analysed?	Build and Unit Test	Code Coverage	Component Tests	Stubbed Endpoint Tests	Integration Tests	Create an Arte
22		Yes	Green	Green - > 90%	Grey - not nee	Green - curren	Green - 2 or 3 mins - part of T	Green
25	Components	Yes	Green - stable and	Amber - good c	Green	Green	Amber - shared Linux box acr	Green
27	Contract	Yes	Green - unit tests g	Grey - not need	Grey - not rele	Grey	Grey	Green
29	Library	Yes	Green - stable and	Amber - good c	Grey -not relev	Grey	Grey	Green
31	Reporting	Yes	Amber - initial test	Amber	Grey - n/a	Grey - does no	Red - Need to be able to test	Amber - vol
33		Yes	Green - fast, appro	Green - good co	Grey - not pres	Amber - decen	Amber - tests are set up and t	Green - larg
34	Interfaces	No						
37	Delivery	Yes	Amber - weird sock	Green	Green	Green	Green	Green
52			Green	Green	Amber - a bit u	Grey - no endp	Amber - a pain to maintain; n	Amber - stil
53								

A heat map for the quality of test stages for different subsystems

The purpose of this push for a bottom-up, shared view of tests at different stages was to increase trust between the Dev teams and the QA team, so that we could avoid test duplication. Previously, separate and overlapping sets of tests were maintained by Dev and QA (Conway's Law at work again!) but by collaborating on the existing tests we could remove duplication without increased risk of regression

[153] http://www.growing-object-oriented-software.com/

problems. The collaborative nature of the workshops helped to get buy-in from teams for ownership of their newly-assigned products and components, and provided useful insights into the health of the code and tests across all the systems we examined.

Increase the focus on operational concerns

The Dev teams had become accustomed to prioritising operational features *below* user-visible features, leaving deployment, diagnostics, metrics, logging, etc. somewhat neglected. This had led to the Ops teams often distrusting releases because their needs had not been addressed. It also made the software more difficult to work with during a live incident, because the hooks and information that would have been useful to diagnose problems were often not present or difficult to find.

We adopted a multi-pronged approach to improving operability:

1. Add version information in DLLs and packages for discoverability and audit
2. Push for log aggregation for both developers and operations people
3. Encourage Dev and Ops collaboration through Run Books

Version-stamping DLLs

A useful feature of .NET DLLs is that the Windows operating system recognises embedded metadata within the files, which allowed us to store useful information within the DLL itself to be discovered later on after deployment. After a hotfix copied new DLLs over existing DLLs the IT Support team could not detect where files had come from. By embedding the Subversion repository revision or a Git SHA in the DLL we had a way to trace back a particular DLL to a specific commit in version control.

Log aggregation

My colleague put in place some cunning build scripts to generate a SCOM[154] management pack, which defined the alerting thresholds for different events in the logs. We defined some sensible defaults for the location of log files on disk, as the logging location had previously been arbitrary[155].

We pushed hard for log aggregation as a first-class capability. We were able to take advantage of a Security audit requirement that IT Support should not remote onto Production machines, with log aggregation recognised as a good solution. We were able to get agreement to make the log search available from tech team machines[156], and also began to demonstrate the benefits of log aggregation for developers using ElasticSearch/LogStash/Kibana (ELK).

In a pilot run by one Dev team, the time taken for diagnosing deployment and runtime failures in their test environment was greatly reduced, and much developer excitement ensued: rather than having to remote onto each of five or six boxes, they could simply use Kibana from their browser to search across logs from all machines. After seeing this, I realised that using ELK in both Operations and Development produces an excellent opportunity for direct, practical collaboration between Ops and Dev, especially during a live incident, as both Dev and Ops people would be familiar with the same Lucene search queries in Kibana.

[154] http://technet.microsoft.com/en-gb/library/hh205987.aspx

[155] I must confess to having a bit of an obsession with software operability. Since I first started writing software commercially, I always added application logging as the very first feature; whether it was a desktop data display tool written in Delphi 5 or a document processing engine for a regulatory reporting environment, I found it hugely valuable to have good logging in place as a sound foundation for subsequent development. I found it quite straightforward to articulate the benefits of good application logging.

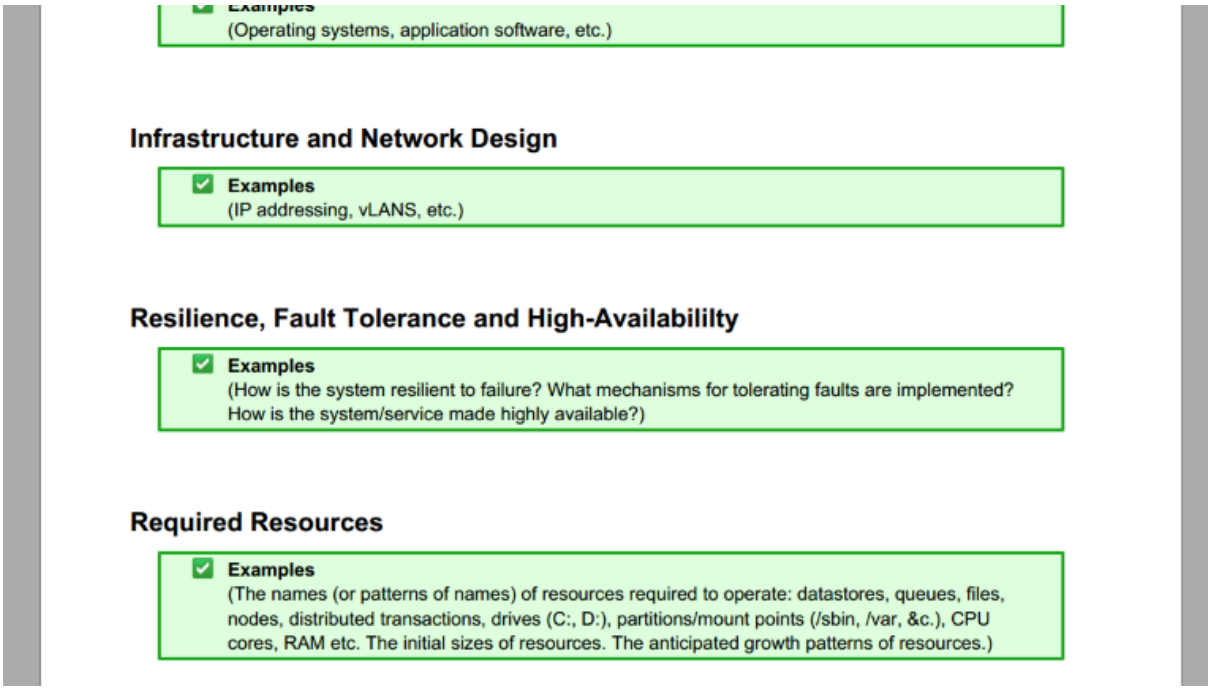
[156] Although the implementation didn't actually begin until after I left.

Run Book templates for an operability checklist

A slow-burn but effective approach to improving operability was to define a common Run Book template covering typical operational concerns and retro-fit these to existing subsystems as well as new systems. The operational criteria covered how applications would address things such as:

- Resilience, Fault Tolerance and High-Availability
- Throttling and Partial Shutdown
- Security and Access Control
- Daylight-saving time changes

Prior to joining the organisation I had found that a focus on operational readiness - although painful for many developers - usually paid dividends after the software went into Production, with reduced errors, enhanced response to failure conditions, and better diagnostic capabilities. This resonated with the IT Service Manager and the Head of IT, and so we began collaborating regularly on the details of the Run Book template as a way to improve how software worked in Production[157].



An extract from an early Run Book template

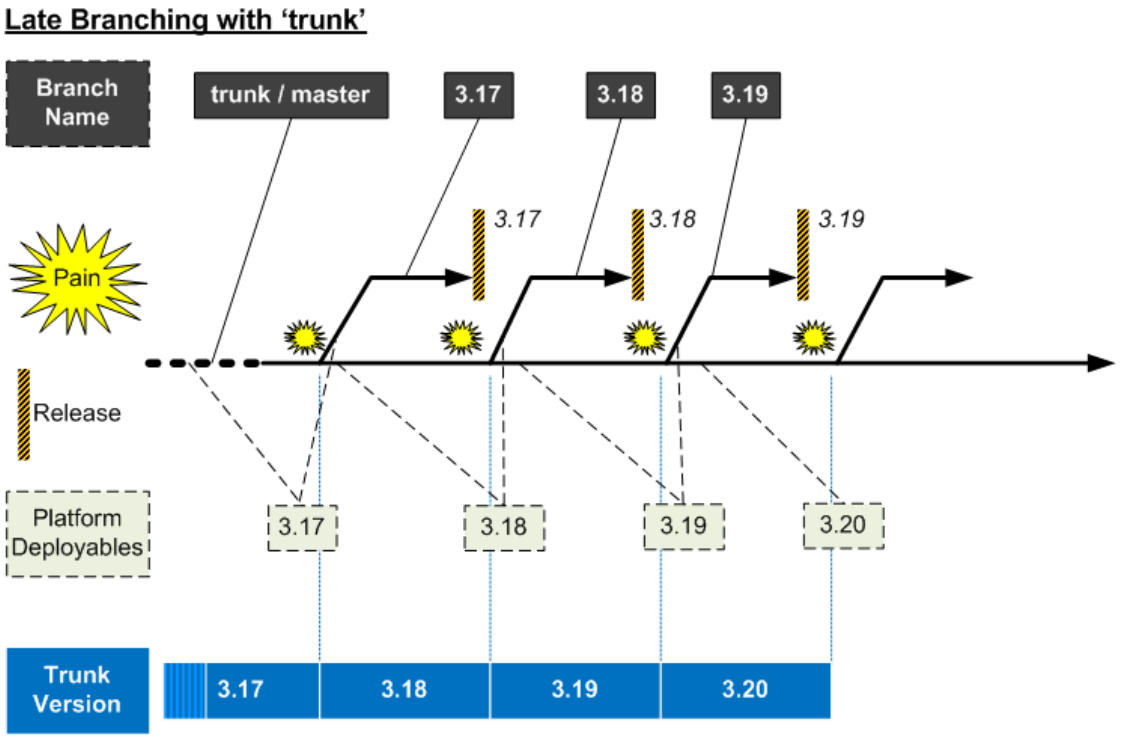
The act of collaborating in this way with key people in IT Operations was in itself useful for building trust between Dev and Ops. We also began introducing the Run Book template to Dev teams working on major new functionality as an operability checklist. If the Dev team was unfamiliar with one of the criteria, IT Operations were on hand to guide the team in some sensible defaults.

Branching is painful

One of the most awkward aspects of platform releases was that everything was released together at the same time. Originally, this had been a fairly sound design decision for the UI part of the codebase, but by 2011 almost all the software components and services were versioned in lockstep. Because the Regression

[157] http://blog.softwareoperability.com/2013/10/16/operability-can-improve-if-developers-write-a-draft-run-book/

Testing and pre-release activities took at least 6 weeks per release, and work for release *N+2* overlapped with previous work for release *N+1* and *N*, we had up to three active source code branches at any one time. The branch names (*3.14*, *3.15*, *3.16*, etc.) had to be created in separate Subversion repositories, and because Subversion was also used for artifacts storage the branch names had to match artifact repositories, too. To make matters worse, development began one release in trunk and then switched to a named branch just before a release was cut:

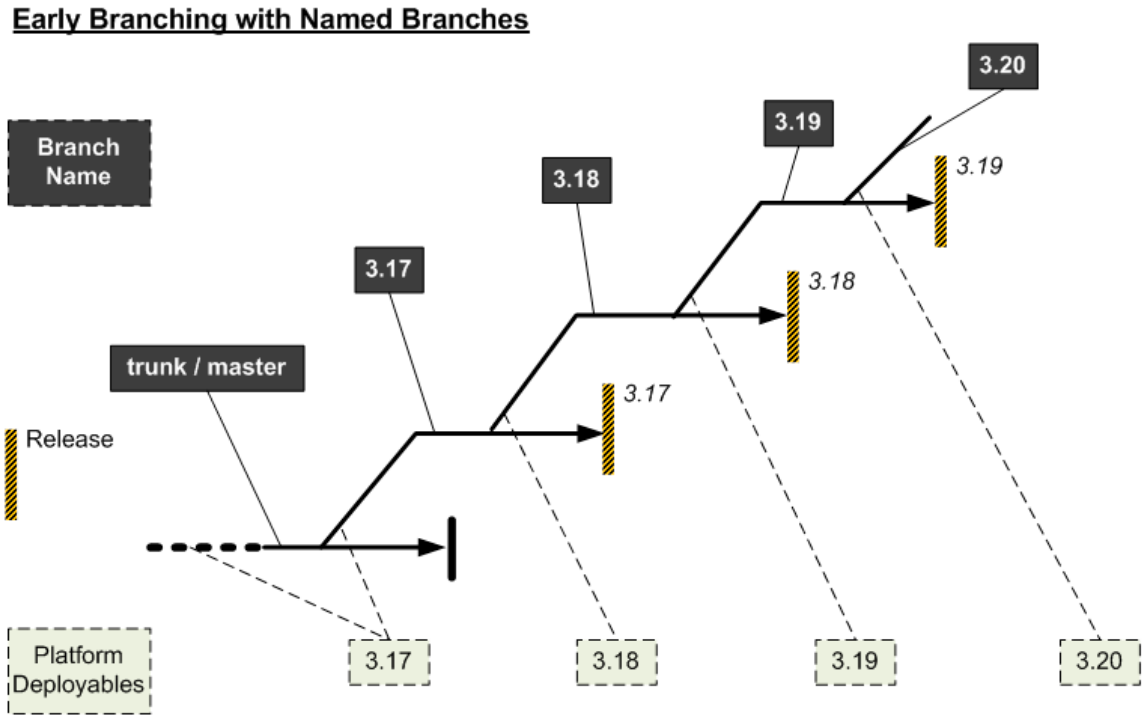


Late branching with trunk - painful

This scheme combined the horrible inefficiencies of branch-based development with the pain that arises if there is a lack of discipline in a trunk-based approach. Development work often halted for several days around the branching activity, with build pipelines going red and managers putting pressure on the Dev teams in London and Bangalore to finish the branching. Work was regularly lost on the wrong branch, and had to be back-ported or removed from future branches.

We wanted to move to Trunk Based Development, but sensed it would take several years to achieve due to the difficulty of breaking out of commitments already given to external customers for feature X in release Y. Trunk Based Development was also made difficult by an internal rigid date-based release pattern, with release engineers staying awake for many nights at a time preparing and testing the release in Pre-Production.

We decided to remove the biggest source of noise from the platform branching scheme by moving the branching date to the start of each development cycle and using fixed branch names:



Early branching with named branches - less painful

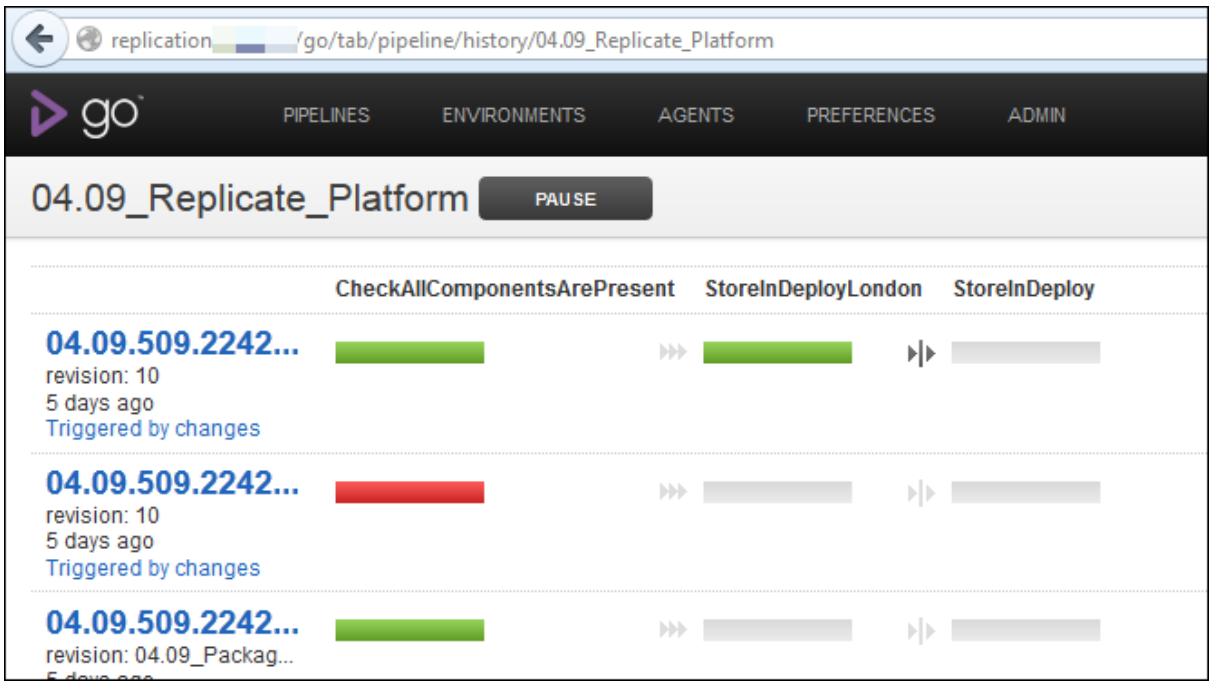
This had the effect of moving away from the spurious trunk-based scheme to a staircase scheme[158] that reduced the confusion and pressure around branching, allowing us to automate most of the branching activities using Chef. Because the branching became more deterministic, we also could push responsibility for getting branch builds green to the Dev teams, and over time this helped to align teams with certain products and parts of the system.

During 2013 we began to remove some software components from the platform branching scheme, giving them their own versioning scheme based on semantic versioning[159]. This allowed teams to communicate intent through version numbering, especially when a new version contained breaking changes. Eventually we reached a point where a platform release was essentially a collection of metadata representing the version numbers of the different components & services to be released. We even moved to a declarative verification model where platform aggregation scripts would fail fast if expected components were deemed invalid:

[158] close to the 'Continually Cascading Codelines' scheme in Streamed Lines.
[159] http://semver.org/

Verifying the platform release aggregation

By using Go to orchestrate the replication and aggregation tasks, we could make visible the work in progress (WIP):



Making replication WIP visible

Over the course of 2 years we reduced branching activity time from 2 weeks to 2 days; when you consider up to 10 Dev teams plus release teams and testers would block on branch completion, this represented a significant cost saving. It was also satisfying also to see improved collaboration across teams as a result of the new branching scheme.

The artifact repository as a first-class service

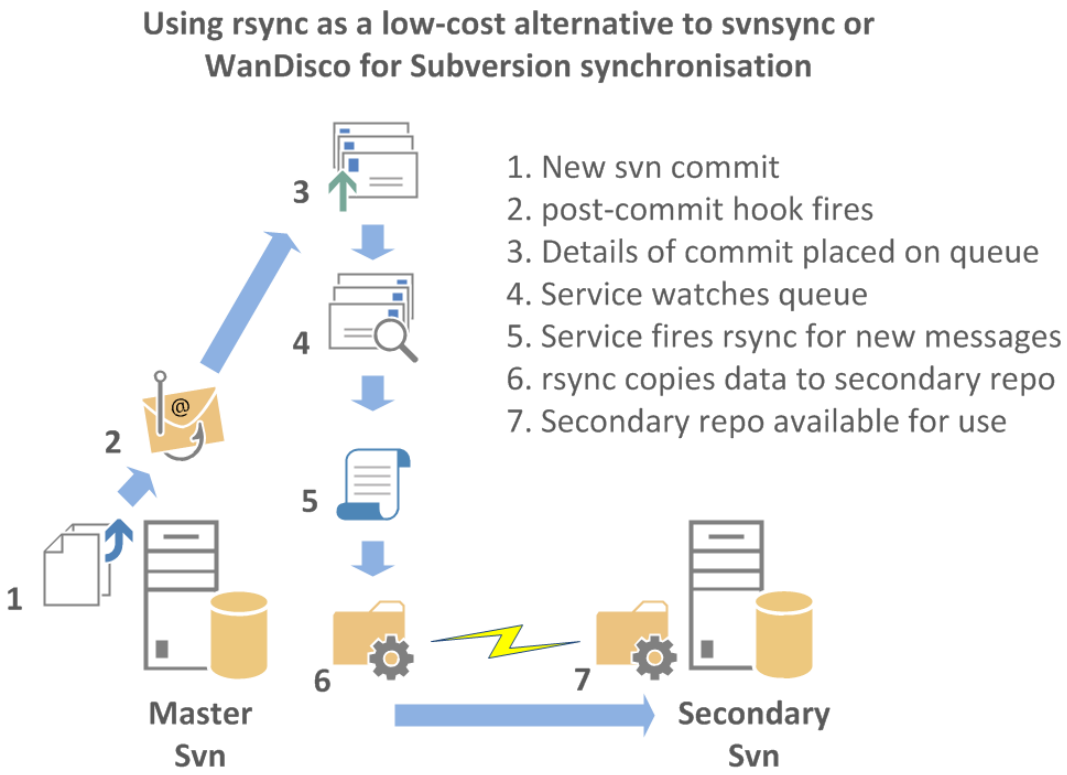
A particularly troublesome part of the initial BDR infrastructure was the replication of source code and artifacts between London and Bangalore. Each location had a build farm which built separate sets of

artifacts which were then combined during integration in London. The flow of materials between the two locations was very complicated and tortuous, using a combination of Git, Subversion, and *svnsync* for replication and duplication of source code and binary artifacts.

Initially it took over eight hours to replicate all the latest artifacts from Bangalore to London, so if a new build was needed we would often wait up to two days for a new set of artifacts owing to the time difference between the UK and India.

Replication represented one of our biggest and most costly delays. Over the course of 2 years, we managed to reduce replication time from 8 hours to 40 minutes for the main set of artifacts[160]. This significantly improved the experience for everyone involved in the build and release of software: dev teams, QA people, and release engineers, reducing waste and saving money. We achieved this through a combination of:

- Fixing configuration problems with the MPLS link between Bangalore and London (at one point it was rate-limiting to 32KB/s per TCP connection)
- Replacing unreliable `svnsync` replication with a home-grown Subversion replication scheme based on rsync
- Running a customised version of NuGet Gallery with Squid caching build artifacts
- Moving more of the CI builds to London so we had to ship only source code
- Splitting up the builds for Bangalore-based components so that artifacts should be shipped independently



Home-grown Subversion replication

[160] in fact, as we split up the platform into more independent pieces, and especially as we moved to Artifactory, the replication time became seconds rather than minutes for individual subsystems

We eventually began moving to Artifactory[161] for storing and replicating artifacts. Artifactory was useful not only for multi-site replication of deployable artifacts, but also for its native package repository support: NuGet for .NET compile-time packages; Chocolatey for .NET run-time packages; and Yum for RPM packages. We found that because Artifactory can be set up to replicate only artifacts that are actually needed in an environment, the volume of replication traffic was much lower, as was the time taken for replication.

Fostering communication, collaboration, and trust

Within the Build & Deployment group in London we had to interact with many different teams: the IT Support team for hardware-related issues with development infrastructure; the Release team for Production-focussed replication; and IT Security for firewall changes. We therefore set up regular meetings with these teams and fervently stuck to the schedule even if we had nothing particular to discuss. We had weekly informal sessions with:

- The Build & Deployment team in Bangalore
- The internal IT Support team
- The Environments team
- The Release team
- Incident Management & Security
- Software architecture
- The QA team (although this was patchy/occasional)

The regularity of meetings was important, as it provided a rhythmic heartbeat for changes and improvements. Small problems were caught and addressed early on, before they became larger problems. In particular, the meetings helped to coordinate activities across different teams, which at the time often had very different drivers or success criteria. Regular sessions meant that people knew that the awkward stuff would be addressed at least once a week, so additional quick everyday discussions were based on higher trust and typically did not expand or go off-topic.

Raising awareness within teams and across the organisation

Team engagement

When I started in 2011, the Dev teams were separated physically from the Ops teams at opposite ends of the building with a cafe/seating area in the middle - a true Dev/Ops divide! I called this out on my first day, and we continued to try to find ways to break down this unhelpful split. Some things were small and simple, like a cheery Yuletide greeting:

[161] http://www.jfrog.com/artifactory/

Ops problem now

Other things were trivial: we took the technical books out of a storage cupboard where they were effectively hidden and put them on display in the open-plan area we used for tech talks in order to encourage people to borrow the books. We also devised and ran training courses in version control fundamentals for Ops people who had never used it before.

More fundamentally, we regularly presented and sought feedback at a weekly lunchtime pizza session, and encouraged people from IT Ops to present. By late 2013 the Incident Manager was doing a monthly report on recent live incidents, explaining what went wrong and how developers could help prevent a repeat incident in future.

Engineering Day

Early on it was clear that many teams did not understand the scale of the technical challenges ahead; for example, at one early meeting the idea of Infrastructure As Code[162] was ridiculed by a fairly senior technologist[163]. To tackle this we conceived an internal technology day for all of the IT department, but as we started to explore talk topics we realised there was a great opportunity to widen event participation by opening it up to teams outside of IT. By presenting technical material in an easy-to-understand way to people in Commercial, Finance, Legal, Marketing, HR, etc., we would have the opportunity to win wider support for the changes we knew were needed for the next five years.

The first such 'Engineering Day' was held in September 2012 in London. There were 16 sessions of around 15-20 minutes each, with presenters from almost every team within the IT department. We had spontaneous cheering from non-IT attendees as the database team showed a query which took over 6 minutes on the old database took only 5 seconds on the new database, we had cloud computing explained by the lead software architect, and a video-link to the office in Bangalore for a session on build system monitoring.

We heard from the Commercial team, who worked closely with front-end developers to build a mobile-friendly application, and demonstrations of how both Fisheye and Graphite could be used by tech teams and non-tech teams for business metrics. The IT Support team ran a tour of the comms room, and our

[162] http://www.jedi.be/blog/2013/05/24/Infrastructure%20as%20Code/
[163] he later left the organisation.

team ran two workshops for complete beginners on HTML and websites with WordPress[164], which gave some attendees the sense of having super powers ("I programmed the web!").

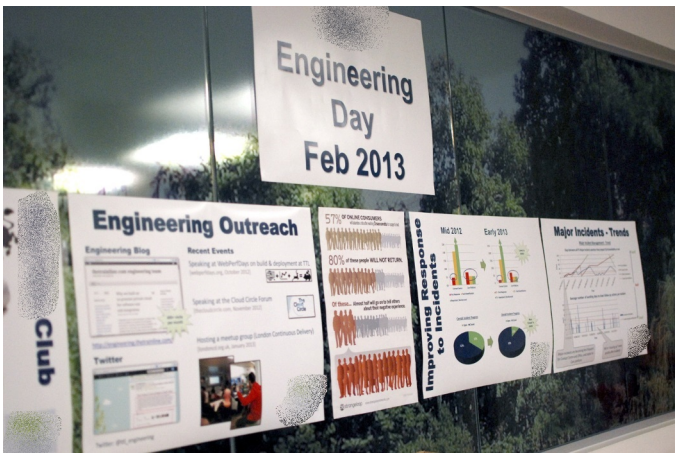
In all, the effect of Engineering Day on teams outside IT was strong:

> *Mahoosive thank-you to [IT] for the Engineering Day. We definitely learnt a lot and appreciate the effort put in to educate the upstairs folk on the technical shizzle going on downstairs. You are all heroes*

We went on to run one of these tech events every six months for a half-day. Through these events, we almost forced a kind of collaboration between the different groups within IT, because on the day we had to present a unified message to the rest of the organisation, which meant finding common ground beforehand. It was great to see people from different teams speaking for perhaps the first time in front of a large group, and people across the organisation offered help with putting on the events.

By October 2013, the events had become a core part of the communications strategy for the IT department, and a good heartbeat for capturing and reflecting on successes and failures during the preceding six months. To 'sell' the idea of these events within your own organisation, I would emphasise that the events:

1. Increase cross-team awareness and collaboration within IT
2. Help to bridge the gap between IT and commercial/Programme teams
3. Begin to increase faith in IT's ability to communicate, to care, and to contribute



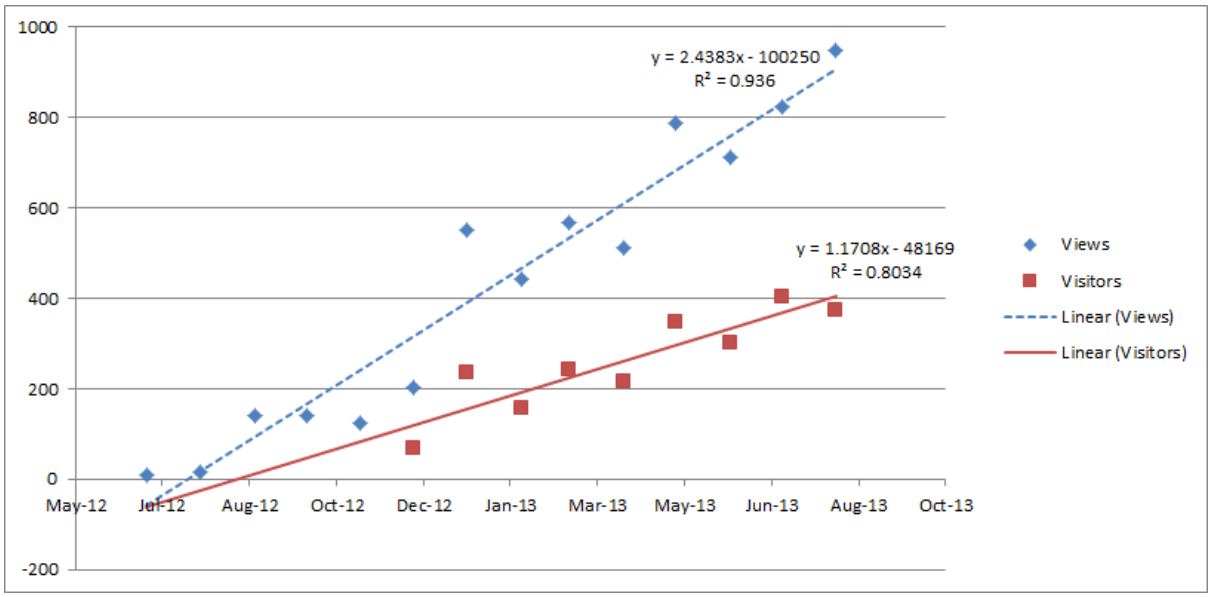
Reach out to the external tech community

Tech blog

One of the key differentiators for many tech-savvy organisations is their public presence: blogs, open source contributions, attendance and speaking at events, sponsoring of meetups and conferences, and interaction with the tech community in general. To help bootstrap this outreach activity, we started a public tech (engineering) blog covering all areas of IT at the organisation: software development, testing, web operations, performance, networks, etc. We worked with the People team (HR) on the messaging

[164] http://blog.matthewskelton.net/2013/03/26/tutorial-how-to-build-your-own-website-using-html-and-wordpress/

and integration with the jobs website; in fact, the main financial justification for the commercial blogging account was a reduced cost of acquiring new tech staff.



A year after the launch in September 2012 we had a modest but steadily increasing 1000 page views per month, and the blog was driving around 20 people per month to the jobs website.

Meetups and conferences

Members of the tech teams also began speaking at (and hosting) meetup groups and conferences. After speaking at WebPerfDays 2012[165] on how software architecture should be a function of build & deployment concerns[166], I met several fellow Continuous Delivery enthusiasts and we decided to start the London Continuous Delivery meetup group[167]; we hosted the first session with an invited speaker at the London offices of the organisation, helping to bootstrap our engineering outreach efforts.

Meetup group at the London offices

People also began blogging about things like attending Velocity Conference, tech books such as *Patterns for Performance and Operability*, and moving the non-core applications to Continuous Delivery.

[165] http://www.webperfdays.org/events/2012-london/index.html
[166] http://www.slideshare.net/matthewskelton/how-build-and-deploy-affectsarchitectureweb-perfdays
[167] http://www.londoncd.org.uk

Results

During my time at the organisation, we in the Build & Deployment teams in London and Bangalore - together with help from many other people - made some significant changes to the way in which the software systems are built. We introduced and improved many different tools to provide enhanced capabilities:

- Git instead of Subversion for source code
- the wider use of version control in general
- Gitolite for multi-site code replication
- Fisheye for code review
- Chef for infrastructure configuration
- Vagrant for VM template changes and infrastructure development
- Artifactory as a first-class artifacts store
- Chocolatey as a .NET package management solution
- LogStash as a log aggregation system for rapid diagnostics
- Linux as the appropriate operating system (rather than Windows) for much of the auxiliary tooling
- Graphite for time-series metrics
- CI pipelines for infrastructure code
- dissemination of the concept of 'infrastructure as code' itself
- deployment pipelines that extend from code commit all the way to Production and visible to all

We also saw our lightweight approach to the operation of internal systems using JIRA tickets copied and adapted by many other teams in IT, increasing the visibility and awareness of work underway. The ability to show visually that work was increasing faster than we could tackle it helped significantly in winning support for additional team members and funding.

Reducing the time taken for replication and distribution of deployable artifacts from 8 hours to 40 minutes (or much less for some subsystems) was an important improvement, and this significantly improved the workflow for many teams. We played a major role in stabilising the release cadence of the main platform to a regular six weeks (with a lead time of 12 weeks), and demonstrated how parts of the platform could be split off and released separately on a much shorter cycle without compromising reliability. We also provided clarity and sanity on changes to teams and responsibilities - particularly the move to 'product-aligned' teams rather than separate silos - and delivered a pragmatic, achieveable roadmap for build & deployment for the subsequent three or four years.

Ultimately, however, I think it was the way in which we helped to bring together previously disparate teams that was the biggest achievement. The Engineering Day events had a marked positive effect on the relationship between IT and other departments, and within IT we facilitated collaboration between Development, QA, IT Support, Architecture, Incident Management, Operations, and Project Management. At one point in late 2013 an IT project manager said to me that the current wisdom within IT was "if you want your project to succeed, get the Eccentric Ninjas involved"; hyperbole, obviously, but it gives an indication of the extent to which we were joining up different bits of the IT department in our daily collaboration and communication.

I should mention that we did not magically fix everything, not by a long way; a substantial amount of work remained to be done in terms of both technology and teams. However, we put the Build & Deployment capability at the organisation on the right track, laying the ground for Continuous Delivery across a good part of the software systems, and opening up ways in which a DevOps culture could take root and grow.

Lessons learnt

My two years at the organisation were an excellent experience. I had the privilege of working with some amazing colleagues in both London and Bangalore, and I simply could not have done half of what I did without their encouragement and inspiration. In fact, we worked more closely and effectively with our counterparts in Bangalore than with some of the teams in London; I'm convinced that co-location of *purpose* beats co-location of desks every time. Above all, I learnt that a high-performing team can do astonishing things given the right context and autonomy.

Some other things I took from that time are:

- **Find opportunities and excuses for collaboration because it produces serendipitous benefits**
  - There is a need to communicate often, with many people, and in simple terms, even when it feels unnatural or difficult. This can take time to have an effect (months or even years), but the effect can be transformative.
  - Regular communication - even if it seems a bit forced at first - helps to reinforce the connections between teams (rather like Hebbian learning[168] in neural networks). With stronger connections, more trust builds, and more useful work can be done.
- **Non-IT really appreciates IT openness**
  - IT departments have a major opportunity to reach out to and impress other departments through good and well-executed communications
- **Find shared language and terminology to help bring teams together**
  - Map ITIL lifecycle or incident management terminology onto software development phases
  - A focus on software operability can be a way to develop trust between Development and Operations
- **Conway's Law applies not only to application architecture but to auxiliary tooling**
  - Avoid centralised monolithic tooling *unless* shared state is needed (e.g. for version control or infrastructure configuration)
  - Separate teams should probably have separate instances of common tooling (e.g. deployment pipeline tools, monitoring config, etc.) or at least a way to configure and query these tools separately
- **Deployment pipelines should be visible and useful to all**
  - Deployment pipelines are not just for moving artifacts towards Production, but for gaining trust and confidence
  - Make pipelines simple enough for less technical staff to be comfortable using
- **Tools can be used in special ways to enable and facilitate collaboration**
  - The collaboration aspect of the tools is often orthogonal to the main tool purpose
  - The way in which a tool is introduced - and the timescale for introduction - can strongly influence the success or failure of a new initiative

I would hesitate in 2015 and beyond before running build/deployment/replication infrastructure internally unless there is a very good reason to do so. Instead, I would start with established SaaS offerings for version control, deployable artifacts, continuous integration, build and test environments, log aggregation, metrics & monitoring, and (possibly) deployment pipelines. A reasonable driver for running some infrastructure internally might be the per-GB cost of a SaaS product, but before believing that the organisation will 'save money' by running a large build/deployment/replication estate internally, the

[168] http://en.wikipedia.org/wiki/Hebbian_theory

organisation should weigh the SaaS costs against the salaries of a team of people (three? six? nine?) to build and operate such a system 'as a service' - the real cost is not small, and business-critical build systems do not 'run themselves'.

It is clear to me that the drivers of behaviour and technology - and the resulting systems architecture - at the organisation are quite typical of many successful organisations, particularly those organisations that do not see building and operating software systems as their core business; the challenges were certainly not unique to this organisation alone. I have since seen some very similar patterns at clients in many different sectors: tourism; financial data; hotels; betting; legal software; online donations; and media organisations. The threads common to these organisations are:

- Building an internal capability is hard when the organisation has previously seen IT as a provider rather than a partner
- Projects and budgets damage the viability of software systems
- Most people don't understand how crucial build & deployment has become

To address these challenges, we must:

- Treat build and test infrastructure as a live system, and either outsource the headache by using SaaS or invest properly in internal infrastructure and teams to run it
- Use programme-based funding rather than project-driven budgets
- Treat software operability as first-class concern, with a special focus on the ability to deploy and monitor software systems
- Recognise that modern cloud-based systems are very different from the early web infrastructures, and require substantial ongoing investment in a 'value-add' operations capability (either SaaS or in-house). 'NoOps' does not mean 'no operations', and 'DevOps' does not mean developers running Production.

Finally, I realised that some people have a very strange idea of 'communication'. During some difficult negotiations over responsibility for patching the Dev and Test servers, someone within IT Operations said "there is no communication problem on the Ops side: we updated the wiki page several weeks ago". This view of 'communication' as a one-way flow of information (or worse, 'requirements') probably explains most of the communication problems seen in IT departments in many different organisations.

The word 'communication' comes from Latin *communicare* meaning 'to share'. Effective collaboration for developing and operating complex software systems requires sharing: sharing of skills; sharing of ideas; sharing of responsibility; sharing of 'failures'; and sharing of respect.

About the contributor

Matthew Skelton *has been building, deploying, and operating commercial software systems since 1998, including systems for London Stock Exchange, TUI Travel, Virgin Mobile, and MRI brain scanning machines. He is co-founder and Principal Consultant at Skelton Thatcher Consulting, a specialist consultancy that helps its clients to make the changes to technology and teams needed for modern 'cloud' software systems. He focuses on helping organisations to adopt and sustain good practices such as Continuous Delivery, DevOps, aspects of ITIL, and software operability.*

Matthew founded and co-runs the 1000-member London Continuous Delivery meetup group, and instigated the first conference in Europe dedicated to Continuous Delivery, PIPELINE Conference. He is a regular speaker at conferences around the world – speaking on DevOps, Continuous Delivery, and software operability – and co-facilitates the popular Experience DevOps workshop series. He is a Chartered Engineer (CEng) and has an academic background in Cybernetics, Neuroscience, and Music. In his spare time he enjoys trail running and learning classical guitar.

Matthew Skelton

34 days - Steve Smith

Steve on Twitter: @AgileSteveSmith[169] - Steve's blog: AlwaysAgileConsulting[170]

Sky Network Services[171]

Timeline: November 2010 -> November 2013

Sky Network Services (SNS) is the broadband and telephony division of BSkyB, which is the largest pay TV broadcaster in the UK. Based in Central London, SNS was established by BSkyB in 2006 to provide a residential broadband service for Sky customers. SNS manages BSkyB equipment in UK telephone exchanges for over 3 million BSkyB customers, and its Information Systems division (SNS IS) is renowned as a premiere destination for agile developers.

In November 2010, SNS was on the verge of a major technology change. BSkyB was looking to invest heavily in Fibre To The Cabinet (FTTC) broadband to compete with British Telecom (BT) and Virgin, and such a fundamental shift in broadband infrastructure meant far-reaching technology changes.

Introduction

> "We were 2 years into our Continuous Delivery programme and a team was outside the pipeline. When I spoke to their team lead I received a lecture I often gave to management. 'You're fixing the wrong problem', the team lead told me. 'You need to fix the siloed teams in Operations, then worry about the technology'.
>
> 'I can fix this place the easy way or the hard way', I replied. 'The easy way is the top-down way, where you replace my manager's manager's manager with me, and I fold all the siloed teams into cross-functional teams overnight. The hard way is the bottom-up way, where we gradually automate a pipeline that shines a light on our organisational flaws.'
>
> 'You've been doing that for a long time', the team lead pointed out.
>
> I looked at him tiredly. 'Yes', I said. 'Do you have a better idea?'"

Hello! Thanks for buying Build Quality In. Matthew and I have enjoyed sharing our contributors' stories, and you've helped Code Club[172] encourage children to try IT regardless of their gender or ethnicity. That is a Good Thing.

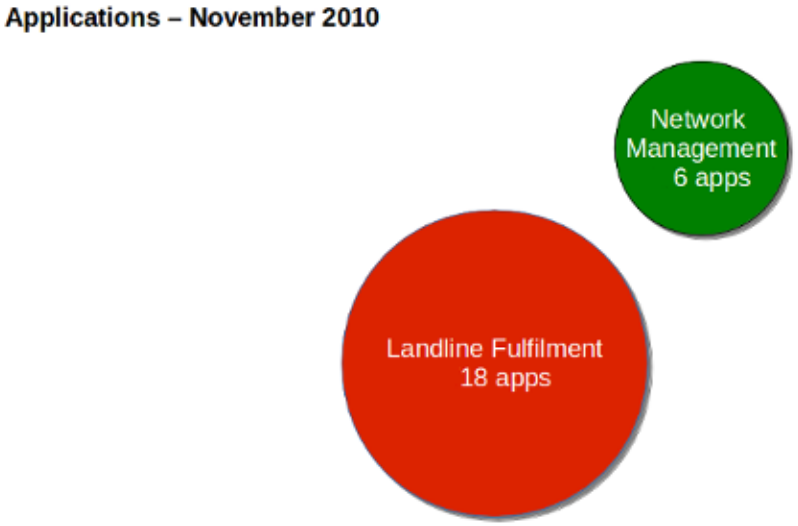
This article bookends our Introduction, which talked about how hard it can be to adopt Continuous Delivery and/or DevOps. While our contributors have provided inspirational success stories - moving from annual to weekly releases, or the badger change manager - it's important we as a community share *all* our stories, as the harder road can also be a rich source of information.

In November 2010 I joined the SNS IS broadband speed checker team as a senior developer. SNS IS is renowned for its commitment to continuously improving its XP practices, with 6 delivery teams of co-located developers, testers, and business analysts practicing 100% pair programming, Test Driven Development, and Acceptance Test Driven Development. It's a great place to code and a pub nearby does a Christmas dinner pie each year. What could be better?

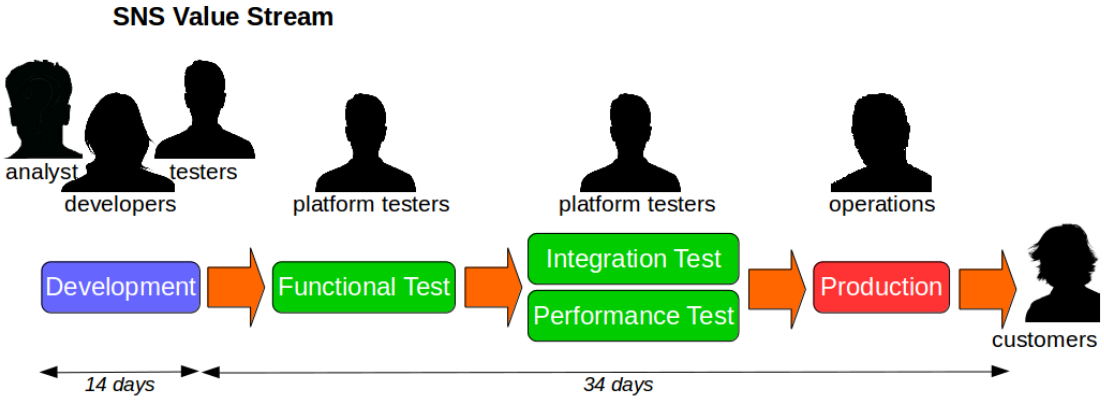
[169] https://twitter.com/AgileSteveSmith
[170] http://www.alwaysagileconsulting.com/blog
[171] http://www.sky.com
[172] http://www.codeclub.org.uk

Year 1

When I joined SNS IS we had 5 teams working on a Landline Fulfilment platform of 18 applications and 1 team working on a Network Management platform of 6 applications. All teams practised Trunk Based Development with a central TeamCity server and Artifactory repository, and all applications were Java WARs run on Tomcat.

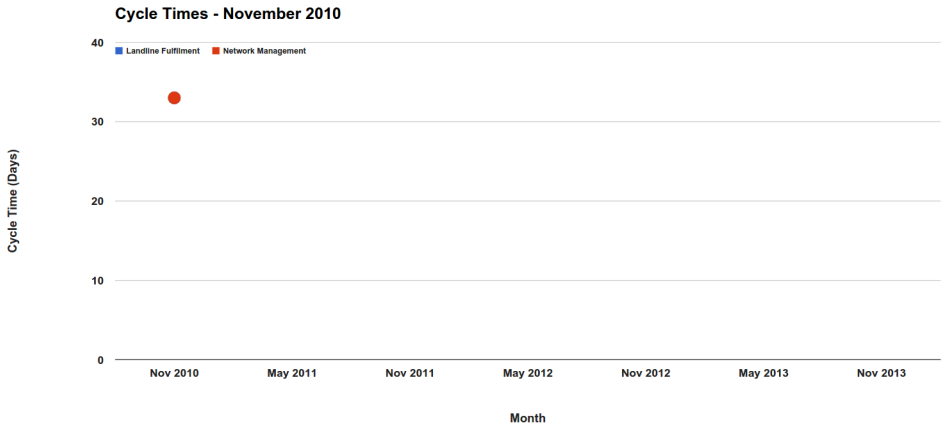


At the end of every other 2 week iteration the delivery teams signed off on a 'major' version of both platforms, and the Release Management team would coordinate releases into multiple test environments for the Platform Test team. Releases were performed by the Operations and Database Administration (DBA) teams, and once Platform Test signed off on a platform version it was released overnight into production on a pre-arranged date.



Discounting ad hoc 'minor' releases for defect fixes, this meant in November 2010 both platforms had a median cycle time[173] of 34 days.

[173]The average lead time from commit to production in a given time period.



As a developer working on Fibre broadband speed checks I was as happy as a pig in poop, but over time I noticed a lot of release problems:

- Development built platforms using a Maven script
- Development deployed/stopped/started an application using a Ruby script
- Operations deployed/stopped/started a platform using a Perl script
- Development migrated an application database using a Maven script
- Operations migrated a platform database using a Perl script
- Development tested an application using automated Java unit/acceptance tests
- Platform Test tested a platform using semi-automated Ruby scripts
- Release Management alone had release visibility via email chain

I soon learned that Development and Operations worked on separate projects with separate priorities, and as a result Conway's Law[174] had wreaked havoc on the release process. Two siloed departments working on the same release process had inevitably resulted in two different toolchains, which meant error rates were high and feedback loops were large[175]. Given my background in Continuous Delivery I found the challenge impossible to resist, and with the support of the Development Manager[176] I unilaterally decided to introduce Continuous Delivery to SNS.

The platform build

In mid-2011 Landline Fulfilment applications were versioned together as non-unique Maven snapshots and rebuilt into fixed version artifacts. The rebuild took 60 minutes and frequently failed, which meant Operations often waited on Development for a platform version. This culminated in an iteration in which Development shamefacedly admitted it could not build Landline Fulfilment - four weeks of features and defect fixes were locked up in unreleasable snapshots due to Maven transitive dependency hell. This strained the relationship between Development and Operations, but I saw it as a crisistunity[177] that could open the door to Continuous Delivery.

I volunteered to improve the platform build, and by replacing Maven with an Ant script I reduced the build time to 10 minutes[178]. Predictable delivery of platform builds re-established trust between Development and Operations, and based on that success I began to talk to people about the benefits of Continuous Delivery.

[174] http://www.melconway.com/Home/Conways_Law.html
[175] I like to imagine Mel Conway sat in an armchair with a brandy, stroking a white cat and laughing hysterically at the IT industry.
[176] I followed him on the train home for a month, vigorously evangelising Continuous Delivery until my station.
[177] http://www.urbandictionary.com/define.php?term=Crisitunity
[178] We eventually got it down to 3 minutes via the Java Zip API.

The artifact container

Soon afterwards I initiated fortnightly pipeline meetings with the Operations team leads to understand their problems, and the Fibre broadband project was proving painful. The existing Perl release script could only handle WARs on Tomcat, and our greenfield applications used newer technologies. We agreed to work towards a single release mechanism that would support old school and new school applications[179].

When Operations and the Development Lead proposed every artifact be wrapped in a zip file with start/stop/status Bash scripts, I saw it as an Artifact Container[180] that could decouple both the pipeline and Operations from application-specific behaviour. While the Development Lead encouraged teams to adopt the interface, I wrote Ant deploy/start/stop scripts to use it. Pushing application-specific knowledge into the applications themselves empowered teams to choose their own technology stack, simplified the release process for Operations, and ensured our fledgling pipeline would scale. It was a big win for us.

The successful small vision

As the Artifact Container began to be adopted, I ran a department meeting optimistically entitled "Continuous Delivery at SNS". I described our release problems, how Continuous Delivery might help, and proposed the following vision:

> *"We can deploy a known good version of an SNS application in a single button click, the same way every time, in every environment"*

My suggested vision was cautiously accepted, in part due to its limited scope. This was a deliberate strategy on my part - our release process was a mess, and a call to arms to reduce cycle time from one developer felt too ambitious. By focussing on a consistent toolchain I hoped to improve release time into test environments and then shine a light on our cycle time constraint.

The aggregate artifacts

While our old school Landline Fulfilment applications were versioned together and released with a single Perl command, our new school applications were versioned and released independently via TeamCity. This meant Operations suddenly had 40+ deployment buttons to use in TeamCity, and errors inevitably crept in[181].

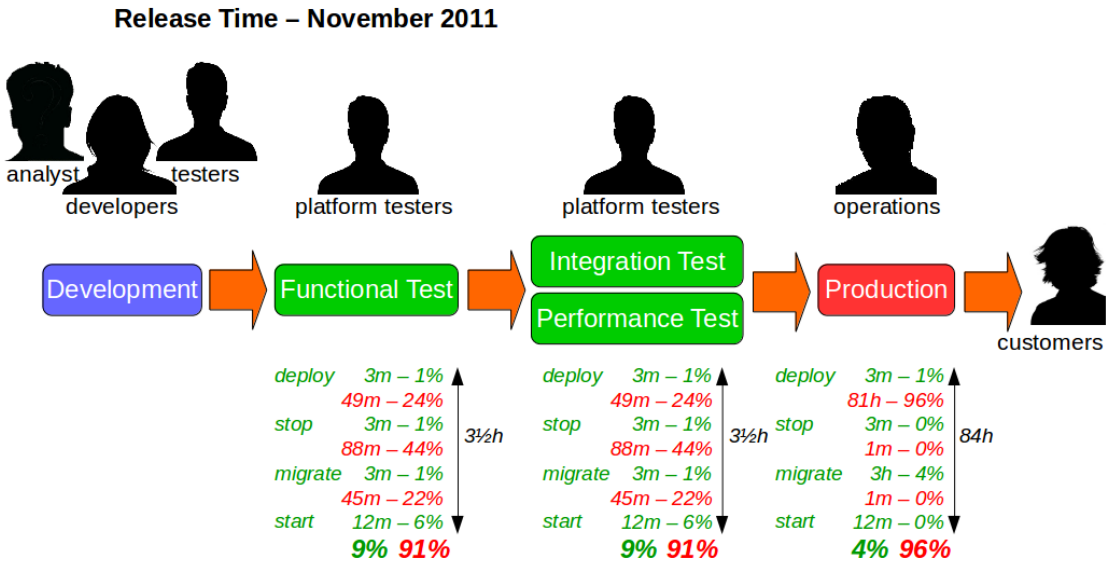
Seeing a platform as a naturally occurring aggregate and its constituents as entities, I created Aggregate Artifacts[182] - versioned XML manifests that described the collection of application versions comprising a platform version. An Aggregate Artifact was created by a user specifying a platform version and a list of pre-built constituent versions, and when an Aggregate Artifact was released into an environment the pipeline released only the new constituents. This prevented artifact duplication, increased visibility of platform versions, and provided an incremental release mechanism.

When our Wifi project started the delivery team immediately used the pipeline, and the combination of Artifact Containers and Aggregate Artifacts worked well. 3 platforms of 11 applications were introduced in quick succession, and people began to ask when old school platforms would be pipelined.

[179] I encouraged the terms 'old school' and 'new school' to avoid negative connotations around legacy applications.
[180] http://www.alwaysagileconsulting.com/pipeline-pattern-artifact-container
[181] Eric Minick and I dubbed this the Deployment Build antipattern.
[182] http://www.alwaysagileconsulting.com/pipeline-pattern-aggregate-artifact

The release time

While Aggregate Artifacts were becoming popular I was scraping wiki pages and emails for release timings[183], and I gradually built up a picture of Landline Fulfilment release time.



I was amazed to find that while test releases and production releases used the same Perl script, the results were entirely different. On average each test release waited 182 of 202 minutes (91%) for DBA and Operations availability, whereas the most recent production release instead waited 81 of 84 (96%) hours for release in an over-use of Decouple Deploy From Release[184].

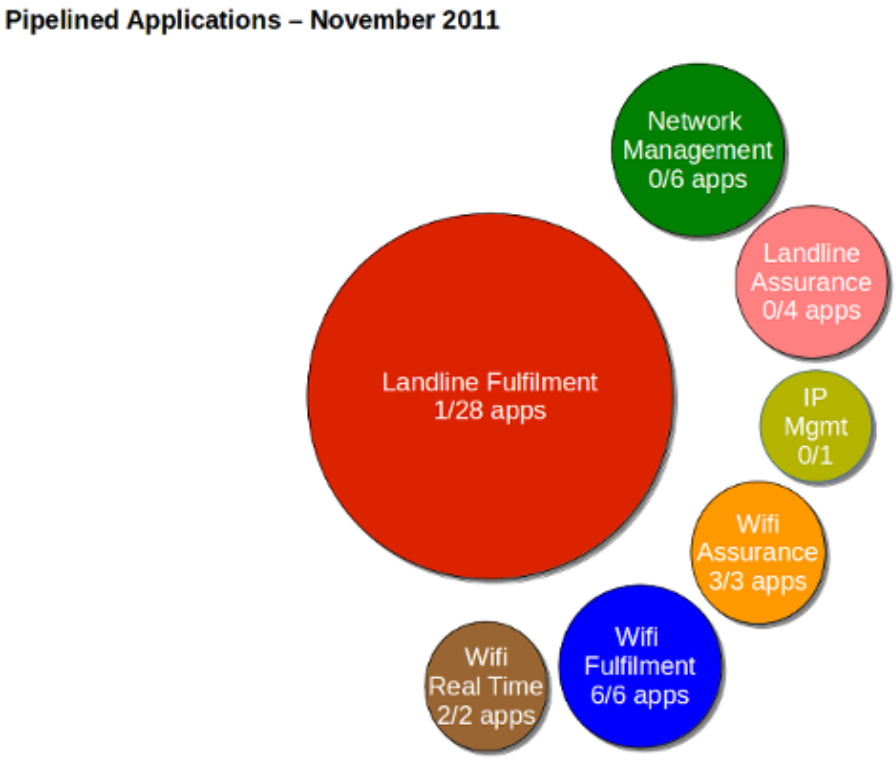
I realised this disparity was because overnight production releases were driven by conversations between people temporarily sat together, whereas daytime test releases were driven by email between people sat in different teams[185]. This implied a push button release performed by Release Management would save 3 hours per test release, and with over 20 monthly test releases that would be another big win.

When I advocated a push button release mechanism Operations were all in favour, but the DBAs were understandably more risk averse[186]. However, the semi-automated DBA Perl scripts could not easily locate new school application SQL, and after several migration failures the DBA team lead agreed to the pipeline deploying all SQL in an Aggregate Artifact to a DBA server. This established a relationship with the DBAs and positioned us for fully automated database migration at a later date[187].

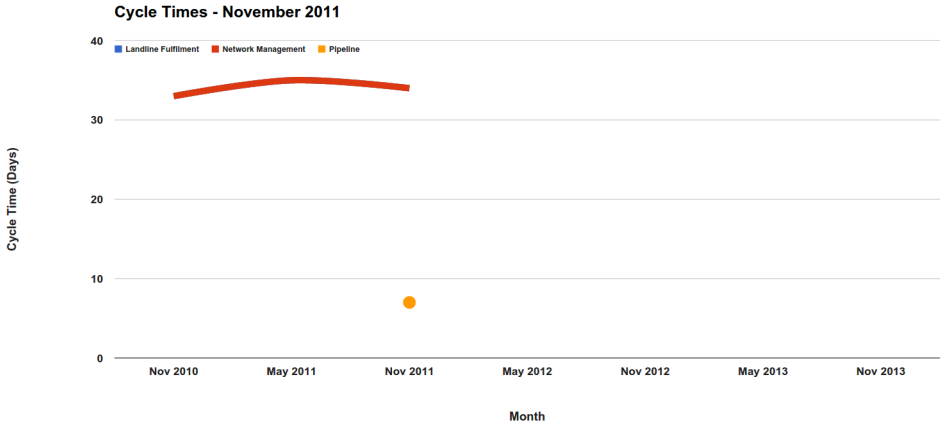
The year end

In November 2011, new school applications were starting to enter the pipeline but old school applications remained outside.

[183] Ah, the glamourous Continuous Delivery lifestyle.
[184] http://www.informit.com/articles/article.aspx?p=1833567&seqNum=2
[185] Damn you, Conway!
[186] You would be risk averse too, if you were given no authority and all the responsibility in the world.
[187] I discovered later on that nobody dared to change the Perl scripts.

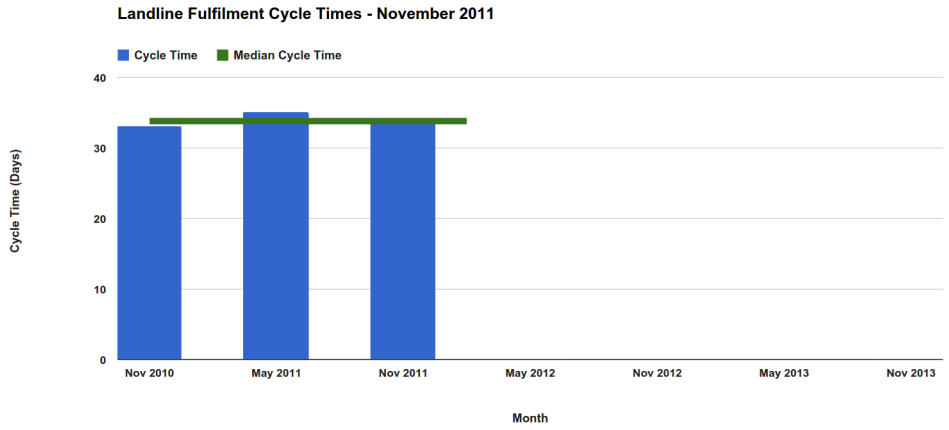


In production, the Landline Fulfilment and Network Management platforms had largely unchanged cycle times. However, the pipeline itself had a low cycle time thanks to a verbal agreement with Release Management that small pipeline releases would be less risky[188].



This meant the median cycle time of the Landline Fulfilment platform I was focussed upon remained 34 days, but with the entire speed checker delivery team now working on the pipeline things were looking up.

[188]Interestingly, people recognised this for centralised infrastructure but not necessarily for business-facing services.



Year 2

The strangler pipeline

While the speed checker team worked on the pipeline, other teams worked on an unprecedented number of new school applications that left us well behind on pipeline integration. Some new applications had even reached production with their own release mechanism, and although Operations wanted one release button per platform the delivery teams were understandably reluctant to discard their hard work.

We overcame this problem by suggesting to delivery teams that we use the Strangler Application pattern[189] to decorate existing release mechanisms. By using Feature Toggles[190] we could switch to application-specific tooling when necessary, and we quickly integrated all remaining Landline Fulfilment applications into the pipeline without any additional risk. This drastically reduced the number of release buttons in TeamCity, and delivery teams were free to discard their own release tooling when they wished[191].

We treated the pipeline like any other application, with its scripts wrapped in an Artifact Container and released by the pipeline[192]. We decoupled the aggregate/environment configuration so teams could change aggregate constituents and/or servers without a pipeline release, but deferred decoupling application configuration as we considered it less valuable.

The snowflake servers

As our application estate grew in size, our infrastructure provisioning model reached breaking point. Our hardware estate was composed of physical Solaris servers manually provisioned by Operations, and as server demand increased there was a spate of pipeline failures caused by Snowflake Server configuration drift.

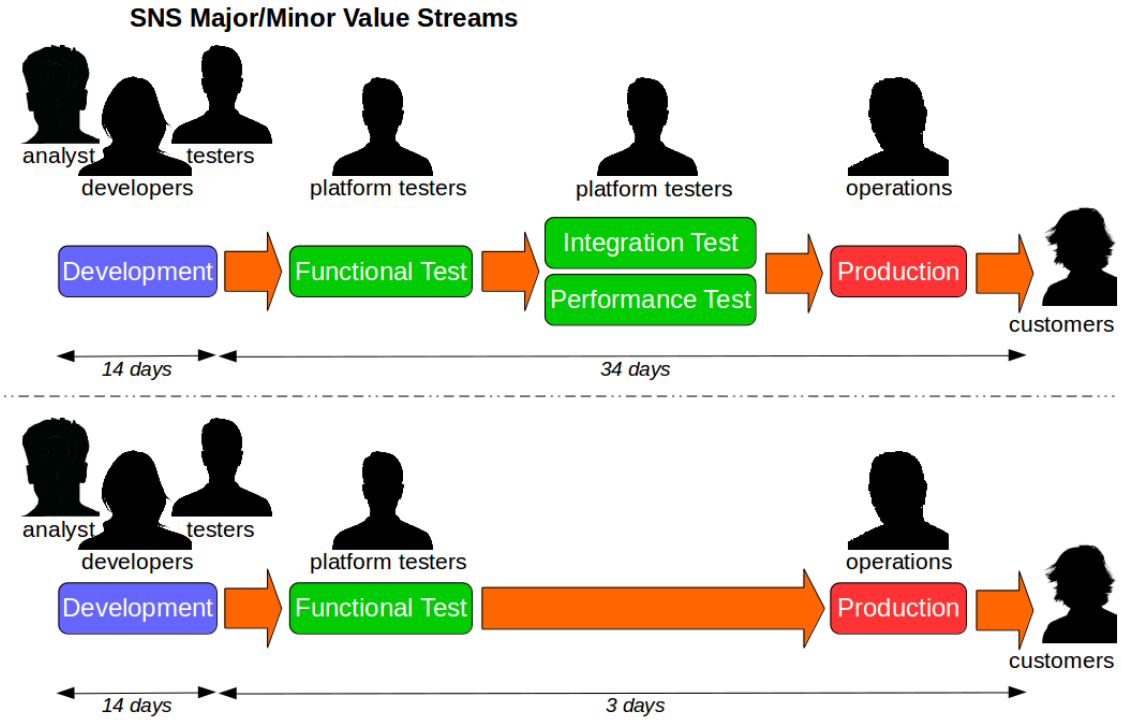
With Operations lacking the time or budget to trial automated infrastructure, we decided to test each server whenever the hardware estate changed. We wrote a script that checked the permissions and directory structure of every server, and triggered it whenever someone added a new server. On its first run the script found over 20 errors, and over the months to come it uncovered many more.

[189] http://www.alwaysagileconsulting.com/pipeline-pattern-stage-stranger/
[190] http://martinfowler.com/bliki/FeatureToggle.html
[191] One team did it a week later, one team two months later, and one team stoically holds out to this day.
[192] Eating your own dog food encourages good practices.

The dashboard

From the outset I had wanted to build a dashboard to radiate release information across the company. While I focussed on pipeline adoption efforts with Operations, the team built an amazing dashboard that showed which application version had been released into which environment. This immediately highlighted the size of the estate - tens of applications were moving through multiple environments at any given time - and it strengthened our argument that further automation was required.

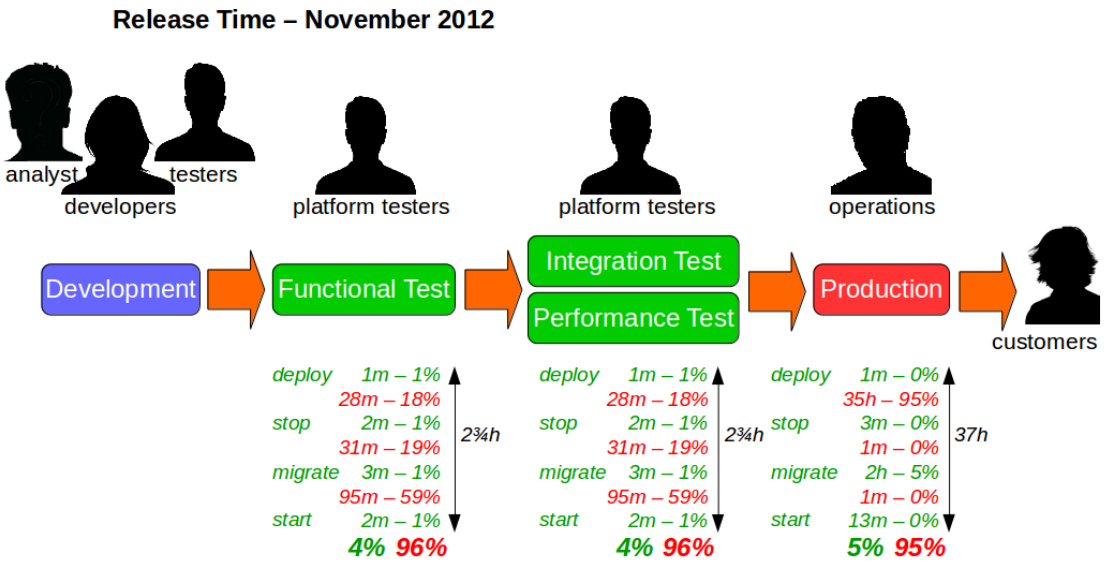
At my insistence the dashboard also visualised a basic value stream mapping, and it provided some fascinating insights into Landline Fulfilment cycle time. Firstly, 'minor' fix releases were more frequent than expected and did not undergo the same testing by Platform Test, resulting in a median cycle time of only 3 days.



The fact we had 2 distinct value streams showed the potential SNS IS had for Continuous Delivery - when the urgency of work was well understood e.g. a production defect, people collaborated to rapidly release a new platform version[193]. It also suggested the testing performed by Platform Test was not always valued, which puzzled me.

The dashboard also provided a release time breakdown for every Landline Fulfilment release, and showed how the pipeline was starting to save us time.

[193] I now call this the Dual Value Streams antipattern.

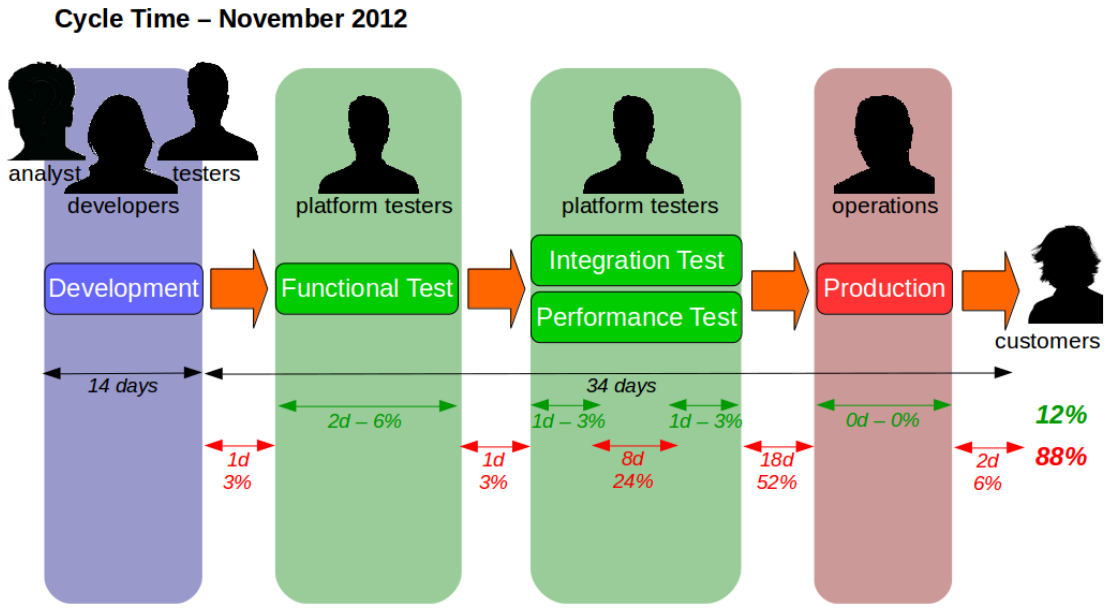


By automating deploy/stop/start actions on the command line and integrating all Landline Fulfilment applications into the pipeline we had reduced release time by 12% since 2011, but there was a long way to go. We advocated adding release buttons into the dashboard so Release Management would be able to self-service releases.

Some releases were infamous outliers. For example, the first release of the Providers platform spent 40 seconds in the pipeline but took 63 days to reach production! The delivery team and Operations were working on orthogonal business projects at the time, which meant hardware procurement had to happen on the critical path. Lead times were inflated, the production launch was delayed, and the delivery team sulked for some time afterwards.

The need for change

As well as documenting release times, the dashboard also showed us where cycle time was spent - and it was eye-opening to say the least.



I had always suspected the majority of our cycle time was unaccounted for, but learning 30 of 34 days was wasted was a sobering moment. It showed our automation efforts were a local optimisation - no matter how good the toolchain, it could not improve the 8 days in Platform Test waiting on a BT integration test or the 18 days waiting for a production release. We clearly had some significant organisational challenges to overcome.

In response to this we made a series of bottom-up attempts to change SNS IS and nudge it towards a new mindset. We evangelised Continuous Delivery in the company language, and listened to the problems of our value stream participants. We identified early adopters in each team, and worked with them on the pipeline to gain early feedback. We formalised a release workflow in the dashboard to reduce release-time errors, and we highlighted the low cycle time rate vs. low defect rate of the pipeline itself.

Changing people is much harder than changing software, and we made plenty of mistakes. We suggested changes to Platform Testing without adequately understanding it. We lost touch with our early adopters over time and feedback dried up as a result. And I famously emailed the DBAs on New Year's Eve to suggest we automate the database dry run to reduce "DBA wait time", and wondered why on the 2nd of January people were cross[194]. The Development Manager and Development Lead were tremendously supportive of our efforts, but at times our mandate was simply too narrow.

The failed big vision

Towards the end of the year, the Development Manager tackled the growing "when will the pipeline finish?" questions[195] by asking me to host a meeting for all team leads and managers to articulate the goals of the pipeline.

On a day's notice, I put everything into it. I emphasised our collective successes - faster Landline Fulfilment builds, Artifact Containers enabling different technologies, Aggregate Artifacts managing the estate, and regular pipeline releases. I described the pipeline as a tool to enable Continuous Delivery, stressed the benefits of releasing smaller changesets more often, and suggested a broader vision:

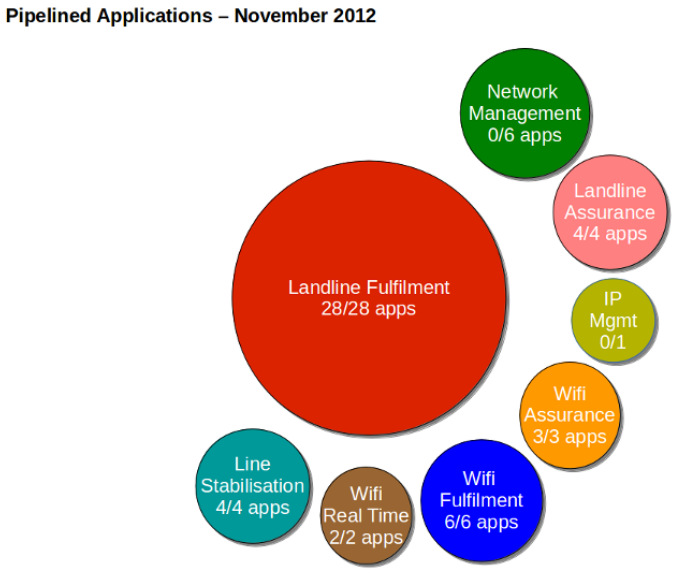
> *Release new versions of applications into production in 2 weeks or less*

[194] "You've ruffled feathers", the Development Manager told me. Don't suggest process changes by email. On New Year's Eve.
[195] Continuous Delivery is a programme, not a project.

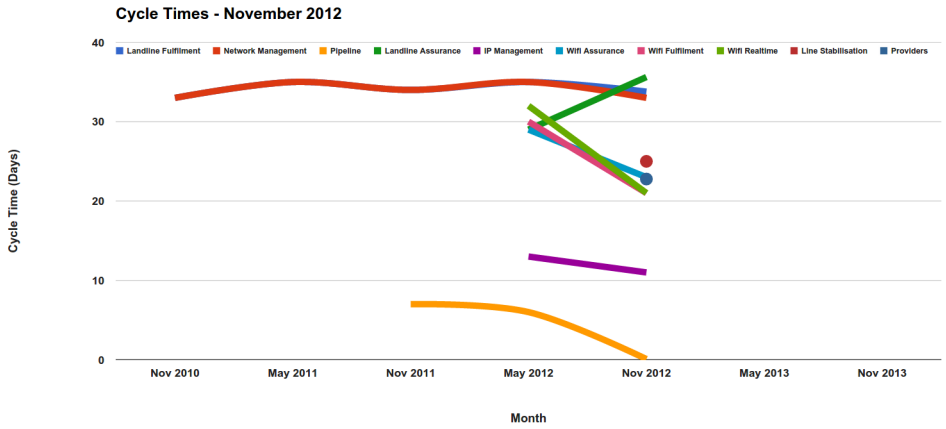
Unfortunately, proposing we release every iteration instead of every other iteration went nowhere. Release Management complained that we had enough trouble releasing every other iteration, the Business Analyst Lead said "there's no incentive from the business to release more frequently", and people disputed whether increasing release cadence would reduce defect risk. It was a missed opportunity.

The year end

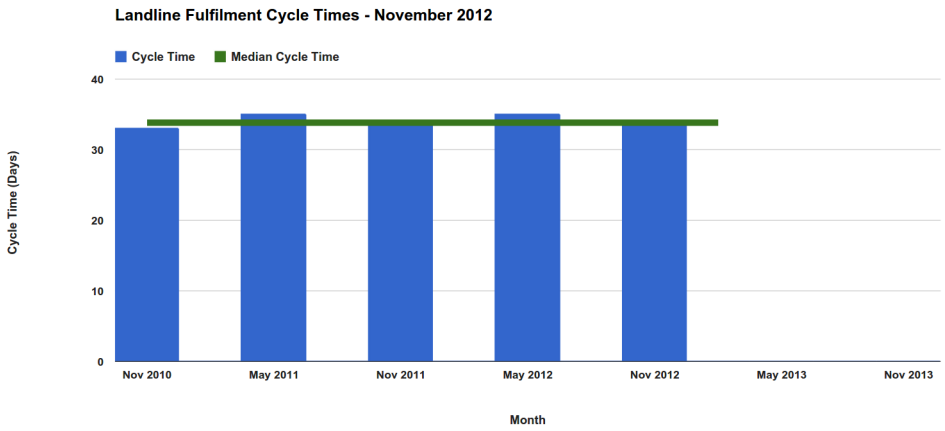
In November 2012, all Landline Fulfilment applications had been integrated into the pipeline, as well as the majority of new school applications.



A number of new school platforms were now in production. The Wifi team had reduced their cycle time and the pipeline itself saw a cycle time reduction from 7 days to 1 day.



However, Landline Fulfilment still had a median cycle time of 34 days and after the failed vision meeting it was unclear how it might be reduced.



Year 3

The coupled releases

On the speed checker team we balanced work between the speed checker and the pipeline, and as 1 of 16 old school Landline Fulfilment applications the speed checker relied on Hessian web services. This meant objects were serialised over the wire[196] and we had to release in lockstep with 15 other applications to prevent runtime incompatibilities. This caused an enormous amount of waste, with only 16 of 64 (25%) of speed checker releases in a six month period actually containing speed checker changes. We needed to independently release the speed checker within the Landline Fulfilment platform, while preserving API compatibility and without incurring additional risk.

We solved this problem by gradually moving the speed checker from Hessian to RESTful XML APIs. As a provider, we offered RESTful endpoints in addition to Hessian and encouraged consumers to migrate as soon as possible. As a consumer we used Verify Branch By Abstraction[197] to incrementally consume REST APIs, and by comparing Hessian and REST results in test environments we uncovered some nasty date/time defects.

We used several techniques to preserve API compatibility between interdependent applications. As a consumer, we used Tolerant Readers[198] to minimise API dependencies. As a provider, we published API Examples[199] derived from our acceptance test data so consumers could unit test our APIs. We also shared Consumer Driven Contracts[200] with other Landline Fulfilment delivery teams, with XPath text files used to unit test consumer expectations.

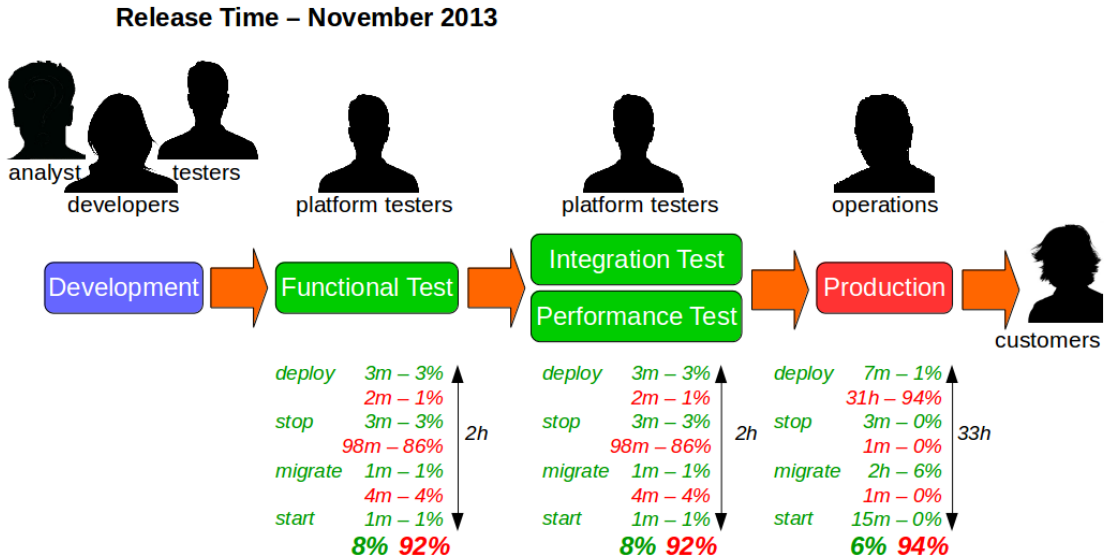
However, mistakes were made and on one particular occasion a team saw fit to pseudo-serialise objects over the wire by reusing JAXB code on either side of a JMS queue[201]. I resorted to Big Stick Ideology[202], and rather than howl in rage I simply plastered their team whiteboard with pictures of kittens until they saw the light.

[196] http://www.alwaysagileconsulting.com/application-antipattern-serialisation
[197] http://www.alwaysagileconsulting.com/application-pattern-verify-branch-by-abstraction
[198] http://martinfowler.com/bliki/TolerantReader.html
[199] http://www.alwaysagileconsulting.com/application-pattern-api-examples
[200] http://www.alwaysagileconsulting.com/application-pattern-consumer-driven-contracts
[201] Don't Repeat Yourself in a bounded context. Do Repeat Yourself between bounded contexts.
[202] https://en.wikipedia.org/wiki/Big_Stick_ideology

[203]

The reduced release time

By mid-2013 we had added pipeline actions into our dashboard and achieved our long-predicted reduction in Landline Fulfilment release time. By including deploy/stop/start buttons the Release Management team were able to assume release responsibility for every platform bar database migration, which meant Operations wait time was eliminated from release time.



With a median release time of 2 hours we had reduced release time by 43% since 2011, which was remarkable given our starting point. However, as DBA wait times were increasing a fully automated database migrator in the pipeline could have produced a 95% reduction and a 10 minute release time.

[203] Photo by author on Meme Generator and used unmodified

The cost of delay

Throughout our Continuous Delivery programme we struggled to prioritise features, and the pipeline backlog meeting often became a bunfight with little appreciation of value[204]. Traditional methods of prioritisation were ineffective as each feature was a one-off infrastructure improvement, and we needed a different approach.

After reading Black Swan Farming[205] I thought we could value features using Cost of Delay[206] - the economic opportunity cost of a feature being available now vs. later - by creating a Cost of Delay model for Continuous Delivery[207] that established a set of benefits by pricing a minute of our time:

1. Reduce Release Time - reduce time to perform a release
2. Avoid Application Release - avoid rework caused by a pipeline failure
3. Avoid Pipeline Release - avoid application release blocking on a pipeline release

We prototyped a Cost of Delay spreadsheet with some crude formulae and a Cost of Delay Divided By Duration[208] (CD3) prioritisation ranking, with an arbitrary minute price of £10,000. We then plugged in our hotly debated features - integrating the Network Management platform, using the pipeline in Landline Fulfilment journey tests, and automating database migration:

```
Reduce Release Time: Network Management deploy/stop/start
CoD = (application wait time / application release cadence) * minute price
CoD = (123 mins / 5 days) * £10,000 = £246,000 per day
CD3 = £246,000 per day / 25 days effort = 9,840

Avoid Application Release: Landline Fulfilment journey tests
CoD = (application release time / application release cadence) * minute price *
failure probability
CoD = (115 mins / 1 day) * £10,000 * 1% = £11,500 per day
CD3 = £11,500 per day / 15 days effort = 767

Reduce Release Time: Landline Fulfilment migration
CoD = (application wait time / application release cadence) * minute price
CoD = (48 mins / 1 day) * £10,000 = £480,000 per day
CD3 = £480,000 per day / 20 days effort = 24,000
```

Description	Benefit	Cost of Delay (£ per day)	Duration (days)	CD3 Priority
Network Management deploy/start/stop	Reduce Release Time	£246,000	25	9,840
Landline Fulfilment journey tests	Avoid Application Release	£11,500	15	767
Landline Fulfilment migration	Reduce Release Time	£480,000	20	24,000

This improved our argument for database migration, but when we plugged in our long-delayed support for independent application definitions we were horrified to find it had been a black swan[209]! The pipeline

[204] Including me, arguing for database migration over the issue du jour.
[205] https://liber.io/v/liberioEpub_53aa4d19e3a7e
[206] http://blackswanfarming.com/cost-of-delay/
[207] http://www.alwaysagileconsulting.com/continuous-delivery-cost-of-delay/
[208] http://blackswanfarming.com/cost-of-delay-divided-by-duration
[209] Remember? That deferred configuration feature over a year ago, that seemed unimportant?

had a cycle time and lead time of 1 day, but the probability of a pipeline change blocking a delivery team was much higher than expected - 23 of 64 (36%) of releases in the past year had contained new applications definitions needed by teams.

```
Avoid Pipeline Release: Independent application definitions
CoD = (pipeline lead time / pipeline cycle time) * minute price * blocking probability
CoD = (1440 mins / 1 day) * £10,000 * 36% = £5,184,000
CD3 = £5,184,000 / 10 days effort = 518,400
```

Description	Benefit	Cost of Delay (£ per day)	Duration (days)	CD3 Priority
Network Management deploy/start/stop	Reduce Release Time	£246,000	25	9,840
Landline Fulfilment journey tests	Avoid Application Release	£11,500	15	767
Landline Fulfilment migration	Reduce Release Time	£480,000	20	24,000
Independent application definitions	Avoid Pipeline Release	£5,184,000	10	518,400

As with other forms of estimation, accuracy was less important than surfacing assumptions and our Cost of Delay prototype showed how bad our prioritisation had been. By underestimating estate growth we had ignored the value of independent application definitions - £5,184,000 per day - for over a year! We increased the priority of that work and looked for other opportunities to use Cost of Delay.

The Release Testing antipattern

Ever since I learned we skipped Platform Testing for 'minor' releases, I had wondered about the Platform Test process. How could it only offer value for 'major' platform releases, and why were our business-critical defects found in production rather than test environments?

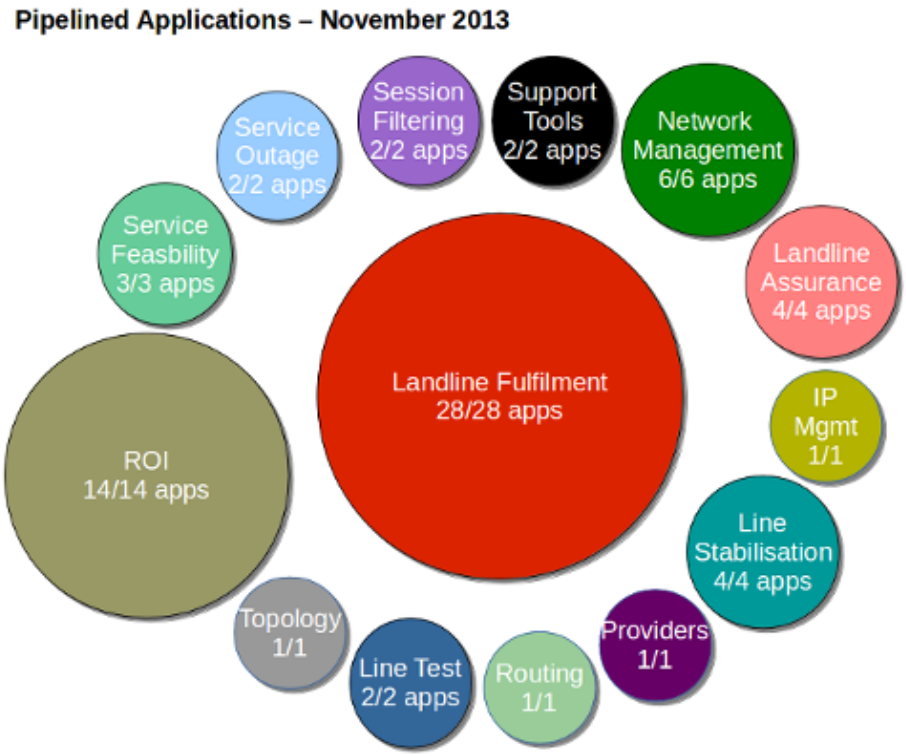
At the end of every other iteration the 4-5 testers in Platform Test received a 'major' version of 2-15 platforms of 3-28 applications, containing 2-4 weeks of features and fixes. They were then given 2 weeks to perform their end-to-end functional, integration, and performance testing prior to production signoff. It was unreasonable to expect anyone to understand that amount of change, and the combination of end-to-end testing and tight deadlines prevented any meaningful test coverage.

Eventually I realised Platform Test was an example of the Release Testing antipattern[210], with the restrictions imposed upon our blameless testers responsible for the production defect count. If we had folded the Platform Test team into the delivery teams we could have invested in exploratory testing and lowered our production defect count, but we could not generate momentum for more organisational change.

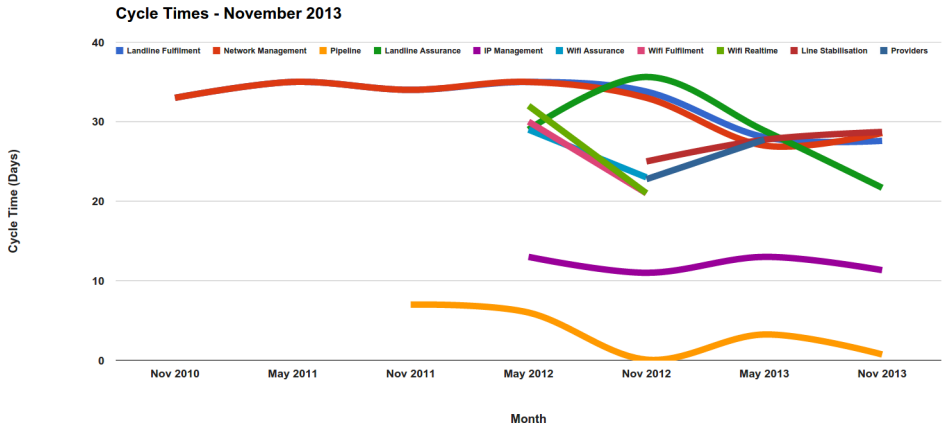
The year end

In November 2013, the entire application estate had been integrated into the pipeline. Enormous credit was due to everyone involved in SNS IS.

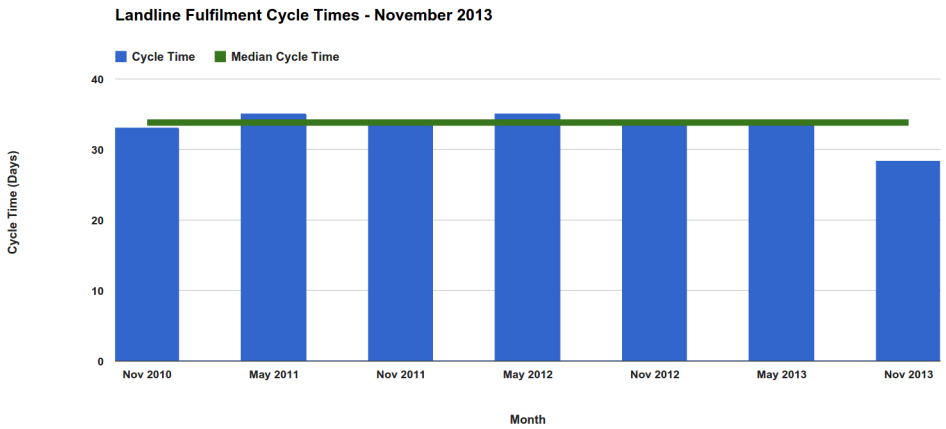
[210] http://www.alwaysagileconsulting.com/organisation-antipattern-release-testing



With the Wifi project unfortunately cancelled the cycle time of old school and new school platforms had congealed around 34 days - a clear indicator that the cycle time constraint was some kind of organisational barrier.



Wait a minute... did you see that? Landline Fulfilment had a lower cycle time in November 2013!



Unfortunately, this had zero impact on the median cycle time of 34 days. Releases in early 2014 gravitated back to the median, and it would have taken another **7 months** of consecutive 28 day cycle times to reduce the median[211].

Reflection

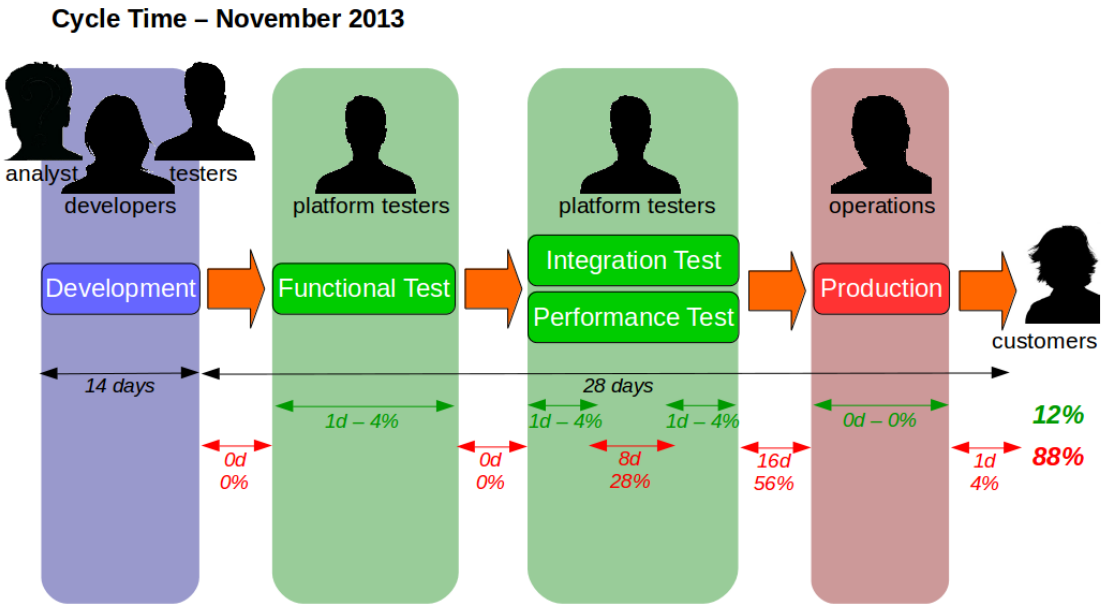
The achievements

The SNS IS Continuous Delivery programme yielded many improvements:

1. Reduced Landline Fulfilment build time by 95% to 3 minutes
2. Built a consistent release mechanism for ~70 applications
3. Introduced a release dashboard visualising all value streams
4. Made a series of organisational changes to improve buy-in and workflow
5. Decoupled Landline Fulfilment applications for independent releases
6. Reduced Landline Fulfilment release time by 43% to 2 hours

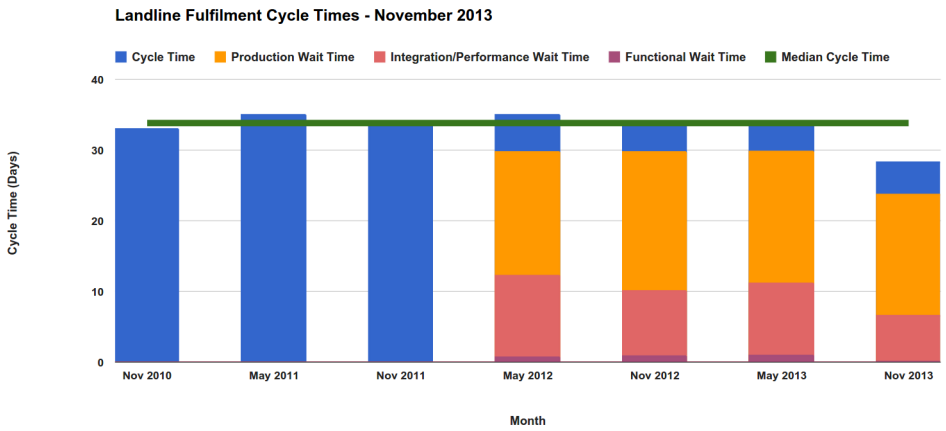
However, Continuous Delivery is all about cycle time and we failed to improve our median cycle time of 34 days. A cycle time breakdown for November 2013 shows little improvement from November 2012.

[211] Not that I'm counting.



The cycle time constraint

A lack of value stream data meant we always struggled to identify our cycle time constraint, with theories ranging from database migration to the Platform Test team. However, 6 months after leaving SNS IS I updated my magic pipeline spreadsheet for this article[212], and by visualising value stream waste vs. cycle time I finally found the answer.



The graph shows 51%-61% of Landline Fulfilment cycle time was production wait time, which means the SNS IS cycle time constraint was BSkyB Change Management - which in hindsight makes complete sense. To release a 'major' version of an application platform into production, SNS IS Release Management had to agree a date with BSkyB Change Management in advance. This was a time-consuming process for Release Management, and as a result tentative production release dates were often scheduled with BSkyB months in advance. This was an understandable tactic but it was a global impediment as *a fixed production date means a fixed cycle time*. While production dates were sometimes delayed by late signoff they were never brought forward on early signoff, so cycle time could increase but not decrease.

[212] I have no life.

While we achieved a temporary cycle time reduction in November 2013 by decreasing Platform Test wait times, we were looking in the wrong place all along. From May 2012 to November 2013, the median wait time between signoff and production release was 17 of 34 days (50%) - which means a re-negotiated process with BSkyB Change Management could have **halved** cycle time without any development effort. During our Continuous Delivery efforts we did not even broach the subject of BSkyB Change Management, and I wish I had thought of this graph 4 years ago.

The lessons learned

What lessons did I learn from our Continuous Delivery journey?

1. Find the business problem. In mid-2013 I locked myself in a New York hotel room with a British sized bagel[213] to answer the SNS IS "no incentive from the business" mentality. When I described my answer[214] to Dave Farley and my original motivation, he simply asked "did you talk to the business?" We should have tracked down the BSkyB broadband product owner and applied Continuous Delivery to their business problems.
2. Map the value stream. It was not until year 2 that our dashboard application visualised the value stream, which meant we flew blind for over a year. We did not understand the delays between teams or the cycle time constraint, which led to uninformed decisions. We should have inserted timings into the old school Perl release scripts and applied Cost of Delay from the outset.
3. Move faster, earlier. As a change agent you are at your most effective when you start. Fewer people know you, fewer people know what you can do, and fewer people will object when you start changing things. I should have pushed harder on key issues such as automated database migration and the role of Platform Test in year 1 when my credibility was at its highest.
4. Get help from others. I had tremendous support from the Development Manager, the Development Lead, and especially the speed checker team. Despite all of the lines of enquiry closed to us - automated infrastructure provisioning, automated database migration, embedded Operations - there was a great camaraderie on the speed checker team that I will never forget.

Looking back, if there is one statement to sum up my journey it would be:

> *Continuous Delivery is 10% automation and 90% organisational change - but don't try it without that 10%*

[213] Me: "Can I have my bagel British-sized, please?" Assistant: "What?" Me: "Er, make it half the normal size and I'll pay the same amount". Assistant (visibly annoyed): "If you want". It was still enormous. America is strange.

[214] http://www.alwaysagileconsulting.com/release-more-with-less

About the contributor

Steve Smith *is an Agile consultant and Continuous Delivery specialist at Always Agile Consulting. An XP/Lean developer with 10+ years of experience in enterprise software development in Britain and New Zealand, Steve favours practices such as Pair Programming and Behaviour Driven Development to build quality into software products. As an early adopter of Continuous Delivery he has overseen transformation programmes in multiple organisations to reduce lead times and increase product revenues. Steve is a co-author of the Continuous Delivery and DevOps book "Build Quality In", a co-organiser of the monthly London Continuous Delivery meetup group, a co-organiser of the annual PIPELINE, and a regular conference speaker.*

Steve Smith

Contributor Q & A

How did you get involved in Continuous Delivery / DevOps?

Chris O'Dell: When I joined 7digital in 2010 the development team there had decided that Continuous Delivery was something they wished to achieve. They'd already made use of a CI server and automated tests but were being held back by the monolithic application. They described the ideal Continuous Delivery environment they planned to achieve, and over the next few years I helped build it with them.

Niek Bartholomeus: I was part of a team that was responsible for the development tools and practices within a large investment bank. at one point we realised that the biggest bottleneck in delivering the software into production was the lack of understanding between the development teams and the operations teams. Until then my activities has remained mostly technical, but this issue really had to be solved on the human level, so a whole different approach was needed.

Rob Lambert and Lyndsay Prewer: We got involved with Continuous Delivery because of a natural evolution of our release process. We needed to get features out to our customers quickly so we had to evolve the release process to facilitate this. It was no good being able to create software rapidly if there was no way to release it to our live environments so we started to optimise our entire build and delivery mechanism.

Phil Wills and Simon Hildrew: Etsy's discussion of how they push to production multiple times a day was one important influence, but a more surprising influence was from many years ago, when we deployed by copying and pasting from one IDE window to another. Whilst the actual process was a mistake-riddled minefield, the the ability to deploy quickly, whenever we needed had been valuable.

Alex Wilson: I was thrown in at the deep end when I started at Unruly with similar experiences to Benji (deploying early on my first day) - since Unruly is my first position after university I believed that this was just how everyone does it, which I discovered is very much not the case.

Benji Weber: I saw how things were done when we joined Unruly, and a team was deploying multiple times a day. I only later discovered it was called Continuous Delivery and read the book by Dave Farley & Jez Humble.

Anna Shipman: GDS is the first place I've worked where they have a DevOps culture so joining GDS was my first exposure. About eighteen months ago I joined the infrastructure team because I wanted to learn how web operations works up close, and that's when I really started to understand the value and purpose of DevOps.

Amy Phillips: Continuous Delivery was one of the options we considered to unblock the Songkick release pipeline. After talking to other companies who were successfully using Continuous Delivery, and Continuous Deployment, we decided it might be the right option for us.

James Betteley: Once upon a time in a software team in London, we needed someone to automate our deployments, and everyone else was out at lunch, so I drew the short straw. It turned out to be more interesting than I'd expected, and learning more about infrastructure and the runtime system was actually good fun. "Infrastructure as code" didn't really exist in those days though, so I've seen the DevOps world emerge and change the way I work immeasurably.

John Clapham: Looking back the first time I got involved was as a Java developer building releases at IngentaConnect, there was a small, bright, team there, and the technical roles overlapped considerably, we just did what was necessary to keep the site healthy. I hadn't experienced alternative approaches and it seemed natural to release when ready, automate a little more each time, and work closely with the people that ran systems. The first time I noticed DevOps as a 'thing', or rather the problem it aimed to solve, was tangentially, through continuous delivery. In 2009 I was working with Nokia Entertainment, where operations and development had evolved away from each other, both technically and culturally. They benefited from optimising in their own domains, but at the expense of cross team collaboration. Working as Product Owner of the continuous delivery Team it soon became clear to us that developing a DevOps culture was key to building a continuous delivery capability.

Jennifer Smith: Working for ThoughtWorks, Continous Delivery is something that has become very much a default setting for me.

The DevOps movement is something that I have followed with interest for some time. As someone that is coming very much more from the 'dev' rather than the 'ops' side I was really interested in understanding more about what goes on underneath the abstractions I work with day to day. Partly through curiosity and wanting to learn new things and partly because I think it's important to have at least an initial understanding as a developer.

Sriram Narayanan: I started off as a developer, and then spent a few years in Infrastructure. I've learned the hard way that I should have invested time in automating various project environments. I learned quite a few lessons and had some great experiences with learning and implement various process and automation improvements. Today, I consult to customers on their CD journey.

Jan-Joost Bouwman: Sometime in 2011 the manager of the IT Service Management department started pushing people to read Continuous Delivery by Jez Humble and David Farley. At first it didn't seem to have a whole lot of relevance to our ITIL based process oriented service management world. But he kept pushing and pushing. So finally I got the book. Coming from a process management background in service management with little development knowledge I didn't get all of it, but I soon understood why my boss was so enthusiastic about this book. I could certainly see the areas where we could use the ideas to improve our performance!

At the same time our development colleagues started experimenting with Agile/Scrum. Soon all development teams were using Agile/Scrum methodologies, but the hand overs to production were still a major obstacle. Yes, we made progress with our quality objectives by standardizing non functional requirements, but it was still a lot of work with loads of paperwork.

When by the end of 2012 the organisational transformation to DevOps was announced I started researching it online and attended some seminars. What I heard there made me realise that the collaboration that comes with DevOps was for us the only way to really unlock the potential of Continuous Delivery. On our internal social network I started our own DevOps community to share what I had learned and read so far, encouraging people to share their stories. Since then the community has grown and grown, reaching well beyond the borders of our own department of 150 DevOps teams to the rest of the ING Netherlands departments and abroad. I still work as a process owner, but have transformed our ITIL processes to the needs of the DevOps Agile/Scrum way of working.

Martin Jackson: I first heard about this "DevOps thing" from a podcast series called Cloud Cafe by John Willis and Michael Cote and thought "Wow I want to be a part of this" and managed to attend my first DevOps days in Hamburg soonish after and it was an eye (and mind) opening experience all those people in the same place with the same problems, working though them together.After that I un-officially got into the world of Continuous Delivery when I worked for Clinical Research company as a Build and Release Engineer, I set up my first Jenkins server and learned through a lot of trials and tribulations that

the path to Continuous Delivery was a bit like trying to cross a minefield some organisations but if you can get there it was well worth it.

Rachel Laycock: Before working at ThoughtWorks I had heard about CD but wasn't really doing it. I joined ThoughtWorks around the time the book came out and there was a real buzz about it in the London office. I didn't really know what it meant so I read the book whilst I was on my first project and immediately became interested in learning more about operations and specifically how to make software operational. This was one of the moments I realised I still had a lot to learn. Those moments still happen pretty often, which is one of the reasons I love my job.

Matthew Skelton: My background is in software development: desktop, mobile, web, and industrial control systems. From around 2006 I became increasingly involved in the build, deployment, and operations of software systems for some large client organisations. It became clear to me that being on Production support is a great way to become a better software developer, and for a while ran a service desk based on ITIL practices. Seeing the effect of addressing (or not addressing) operational concerns as part of software delivery made a big impression on me.

Steve Smith: Between 2008 and 2010 I worked with Dave Farley and a bunch of other smart people at LMAX, where we lovingly crafted a pipeline out of Cruise Control 2.8.2 and Ant 1.7. When I paired with Dave I teased him about the time it was taking him to write his "Continuous Delivery" book, and felt like an utter idiot once it was published.

What do you see as the biggest advantage of Continuous Delivery / DevOps?

Chris: Can I pick two? One technical and one cultural. Technically it's the ability to deliver a flexible system which can respond to changes in direction. This was particularly relevant at 7digital where the music and digital media industry is fast moving and we needed to be able to deliver quick enough to maintain and exceed the pace of the market. Culturally, it brought the whole company closer together. Trust and teamwork greatly increased and so the quality of the entire platform was massively improved as a result. We had a phrase at 7digital – "Use The Voice", which reminded people to talk in person instead of via email or tickets. I believe this a cultural artifact of the faster moving Continuous Delivery approach.

Niek: When the development and operations teams have the possibility to learn from each other, to appreciate each others work, they will start seeing the bigger picture of the work they are doing, which is delivering value to the end user. This insight alone will help them make decisions in their own area that are beneficial for the whole organisation, and not only for their team.

Rob and Lyndsay: The biggest advantage of Continuous Delivery is the ability to release small incremental changes rapidly and then get feedback on these changes. This short feedback loop ensures our customers get the right features quickly and with minimal wasted development time. Features being used in a live environment by real customers is the ultimate feedback as to whether you've built the right thing in the right way and Continuous Delivery helps us to achieve this.

Phil and Simon: Continuous Delivery allows you to build more useful software in a more collaborative, fun way by shortening the feedback cycle.

Alex: CD and automation go hand-in-hand as I see it, and in order to keep our feedback loops tight and enable early decision making we've had to automate our processes aggressively. As an added bonus of this, we're able to quickly deploy servers and applications from scratch with a reduced amount of pain - a victory in its own right.

Benji: It allows us to get feedback on the value of our work in the shortest possible time. Often it is even possible to measure how much money a change is generating/saving after releasing it. Therefore it minimises the amount of time we spend building the wrong things.

Anna: From an organisation's perspective, a DevOps culture underpins the most important parts of an agile software development methodology: release early, release often, fail fast, etc, as well as communication between all parts of the business.

Amy: Being able to release features and bug fixes to the user as soon as they are ready. Moving to Continuous Delivery means we have a release pipeline that aids the team rather than controls it.

From a personal perspective, the biggest advantage for me has been the opportunity to get hands-on experience with web operations. Understanding how your code works in production, how it is supported, and what can go wrong, makes you a much better developer.

James: "The biggest advantage of DevOps is that it reduces your time to market by 40%" Seriously, there is actually a report out there which makes this claim. It's my favourite DevOps made-up fact. In reality, for me the biggest advantage of DevOps is better quality systems. Where a DevOps culture exists, delivery teams are able to design and deliver software which is tailor made for the production environment, which means more reliable, better performing systems.

John: That's a tough question, because when Continuous Delivery and DevOps are done right there are numerous, sometimes unexpected, benefits. I'll start with a caveat though, because it's perfectly possible to get it wrong. For example, investing a disproportionate amount of time or money, or alienating people and reducing engagement with an insensitive approach to change. It is also easy for tool and process choices to have a detrimental impact on quality, reliability and security. The biggest advantage in my mind is that Continuous Delivery is one of those rare initiatives that is good for business and individuals. Done right it accelerates the organisation's ability to learn, assuming it knows what it's goals are and can react to feedback. Releasing changes often means more opportunities to try something, check feedback, refine and try again. Damon Edwards put it nicely: 'You get more shots at the prize'. On the individual side, people working in a Continuous Delivery and DevOps environment are often more engaged, putting responsibly in the right places promotes learning, eases frustration, increases autonomy and encourages personal development.

Jen: I think that the biggest advantage of the DevOps movement/culture is ably demonstrated by the portmanteau itself - bringing together the two perspectives of development and operations together for greater good.

These two groups of people can learn a lot from one another - developers like myself who are exposed to running systems get a really good understanding of the challenges of keeping systems running, available and stable. I can't speak too much from the other perspective, but I feel like there are developer-lead approaches to automation and readability that are a major benefit.

As I mention in my article, exposing developers to operations and support is one step on the way to helping them support their own systems. This not only gives them great experience and feedback but also frees up the operations folks to do bigger and better things.

Sriram: The ability to apply fixes and add features boldly without the constraints on environments, deployment times, long test times, or worries that additions or changes to the code would introduce new defects or re-introduce bugs.

Jan-Joost: For me the biggest advantage is the shared responsibility. There is immense value in letting developers share responsibility for the stability of your production environment. That combined with the immediate feedback from the business at the end (and during) each sprint have been a real boost for our quality.

Martin: The biggest advantage I see is to get the right thing into customer hands in the fastest possible manner possible by doing just enough design while constantly iterating, experimenting and refining. Also the ability to fail fast and correct course quickly allows us to challenge the way we do things without the fear of getting it wrong paralysing us.

Rachel: To me it's about thinking about how software will run in production right from the start. Not just is this feature built well and maintainable from a code design perspective, but how does it need to run, what kind of hardware, caching, networking and operating system. I think it's good for developers to have to learn that early because the "beautiful software" you write doesn't really amount to much until it's being used and is under the load of real users.

Matthew: The biggest advantage for organisations that manage to adopt and sustain a Continuous Delivery or DevOps approach for their software systems is that those systems can be built, tested, deployed, and operated much more effectively than before. The systems can be made more relevant to the organisation's requirements by being easier to change and improve on an ongoing basis, providing better 'return on investment'.

Steve: Releasing smaller changesets more frequently increases revenues and reduces risk. What's not to like?

What do you see as the biggest challenge in Continuous Delivery / DevOps?

Chris: Trust. Trust in your tests, trust in your deployment scripts, trust in the production environment. Trust in the Quality Assurance person, trust in the Product Manager, trust in Sales and Marketing. These things are hard to earn and come with time, repeated proof of robustness, openness and honesty.

Niek: In many traditional enterprises there is still an old-school mindset that software development is not a strategic advantage and therefore cost-efficiency is considered more important than delivering business value. In these enterprises the teams in the IT department are typically structured by technology, effectively creating specialist silo-teams with little opportunity for mutual collaboration, in an attempt to increase efficiency. Unfortunately, these types of organisation structure simply don't work in a domain like software development that is complex and creative. Instead, what is need is a better collaboration and alignment of the objectives between the teams, or, in other words, DevOps.

As this old-school mindset gets stronger as we go up in the hierarchy of the organigram, the easiest strategy on first sight to introduce DevOps seems to be a bottom-up approach where the people on the floor initiate the effort and then let it organically "infect" the rest of the organisation. However, as the old and new mindsets are so fundamentally different it may not be possible to make the switch by gradually evolving from one to the other. Instead we may need something more disruptive, where the pressure to change not only comes from the bottom, but also from the top, and from everywhere else.

In the end, my guess is that only external forces like increasing competition, shrinking market shares and eventually the struggle to survive can set the stage for building up the necessary pressure to change.

Rob and Lyndsay: The biggest challenge in Continuous Delivery is in creating a safe, predictable, efficient and careful way to deploy software in to live with minimal impact on our customer's service. Releasing often means frequent disruptions to your customers if the Continuous Delivery process is not effective so controlling a frequent release process and then optimising it can often be the first challenge when moving to Continuous Delivery.

Phil and Simon: Fear. There's always the temptation to add one more step into the process for getting

software to production when something goes wrong. Sometimes that's the right thing to do, but it's a real cost that should only be paid as a last resort.

Alex: As with any methodology, it can be hard to get buy-in from other developers when the benefits aren't immediately obvious and it's difficult to understand the value of doing CD without having tried it first-hand. It's also easy to fall into the "tooling trap" where you focus too much on tooling and not enough on actual practices - you can easily implement a decent CD workflow with nothing more than a set of shell scripts.

Benji: Like most things in software development, the biggest challenge is communication. Helping new team members understand why we do CD, and how to do it effectively, and maintaining awareness in the team about the reasons for what we do. There are constant forces pulling us away from CD. We add new tests which slow down deploy cycles. We are influenced by hype and "best practices" from the rest of the software development community which don't play well with CD.

Anna: Making sure the concept is understood. As with Agile before it, misunderstandings of DevOps are rife, and also like Agile, people are trying to adopt it, or say they are adopting it, without being clear about the concepts behind it and what the costs as well as the benefits are. The biggest challenge is communicating what DevOps actually means and what is required from the business to support it.

Amy: Creating and maintaining the culture needed to prevent low quality releases. Everyone involved in the delivery needs to understand what you are trying to achieve and needs to allow time for code (and test) design and maintenance.

James: I think it's the whole "cultural change" thing. Most people now "get" that DevOps is about culture (as opposed to being just about automation), but what we still struggle to understand is exactly how to change a culture effectively. Depending on your existing culture, changing to DevOps can be a radical departure, and people just don't appreciate how hard that can be.

John: People; their history and habits. In this area tools have evolved rapidly, there are some amazing projects out there, and smart people to use and integrate them. Ultimately though it's people that start the conversations that lead to technical changes. Despite what appear to be obvious benefits it can take an age for initiatives in Continuous Delivery and DevOps to gain traction. Often it is not a matter of a couple of people experimenting, the change spans departments, challenges existing roles, and the principles and conventions that built careers. There is a wealth of anecdotal and data evidence to support the approach. Despite this some organisations, and people, are still reluctant to take the first steps, the challenge is finding ways to encourage them…

Jen: I think that one of the biggest challenges is around the complexity of the tools, stacks and concepts in the operations space. People who have been doing it a while can start to lose the appreciation of this complexity (maybe a curse of knowledge effect http://en.wikipedia.org/wiki/Curse_of_knowledge).

Although some things just are plain complicated, we can work a lot on addressing this via tools, visualisation and just taking a different perspective on the issue.

Sriram: Lack of executive buy in backed by strong action can be the biggest challenge. You may have the best of technology, developers, intent and automation, showcase all manner of ideas for improvement. But if there isn't executive level buy in backed by concrete action, then the status quo will not change and any initiative can fall flat on its face.

Jan-Joost: For me the biggest advantage is the shared responsibility. There is immense value in letting developers share responsibility for the stability of your production environment. That combined with the immediate feedback from the business at the end (and during) each sprint have been a real boost for our quality.

Martin: I see culture as on of the biggest enablers and obstacles to Continuous Delivery and DevOps without a bottom up culture supported from the top down many Continuous Delivery and DevOps initiatives will literally die on the vine without delivering any benefits and literally poison the well for future initiatives.

Rachel: That's easy, your organisational structure as that informs your design, which in turn informs how it will run.

Matthew: The biggest challenge for organisations wanting to adopt Continuous Delivery and DevOps – a challenge I see time and again with many different client organisations – is that we as IT folk have so far done a very poor job at explaining to non-techies (and other techies) that building and operating modern, complex software systems is a high-skill, value-add activity, not simply the 'execute-only' activity that many people seem to think it is. We need to become better at characterising the patterns of teams, technology choices, and funding models that will help to produce more effective systems without ever-increasing costs due to the accretion of functionality via a naive 'new features first' approach to software evolution.

Steve: Changing organisations to be better aligned with product development flow. Automating a release script is child's play compared to convincing people they should actually talk to each other.

Version History

Version 22 - *22 July 2018*

- Add ISBN details and clarify publisher details

Version 21 - *12th January 2015*

- Minor tweaks for Matthew Skelton and Sriram Narayanan

Version 20 - *6th January 2015*

- Image fixes for Matthew Skelton and Jan-Joost Bouwman

Version 19 - *3rd December 2014*

- Steve Smith

Version 18 - *28th November 2014*

- Matthew Skelton

Version 17 - *21st November 2014*

- Rachel Laycock

Version 16 - *17th November 2014*

- DevOps foreword

Version 15 - *17th November 2014*

- Martin Jackson

Version 14 - *14th November 2014*

- Jan-Joost Bouwman

Version 13 - *12th November 2014*

- Sriram Narayanan

Version 12 - *22nd October 2014*

- Jennifer Smith

Version 11 - *20th October 2014*

- John Clapham

Version 10 - *7th October 2014*

- James Betteley

Version 9 - *5th September 2014*

- Amy Phillips

Version 8 - *2nd September 2014*

- Anna Shipman

Version 7 - *1st September 2014*

- Alex Wilson and Benji Weber

Version 6 - *15th August 2014*

- Marc Cluet

Version 4 - *8th August 2014*

- Phil Wills and Simon Hildrew

Version 3 - *4th August 2014*

- Rob Lambert and Lyndsay Prewer

Version 2 - *1st August 2014*

- Niek Bartholomeus

Version 1 - *29th July 2014*

- Continuous Delivery foreword
- Chris O' Dell

Publisher details

Build Quality In is published by Conflux Digital Ltd - confluxdigital.net[215]

Please direct any queries to publications@confluxdigital.net[216]

[215] https://confluxdigital.net/
[216] mailto:publications@confluxdigital.net