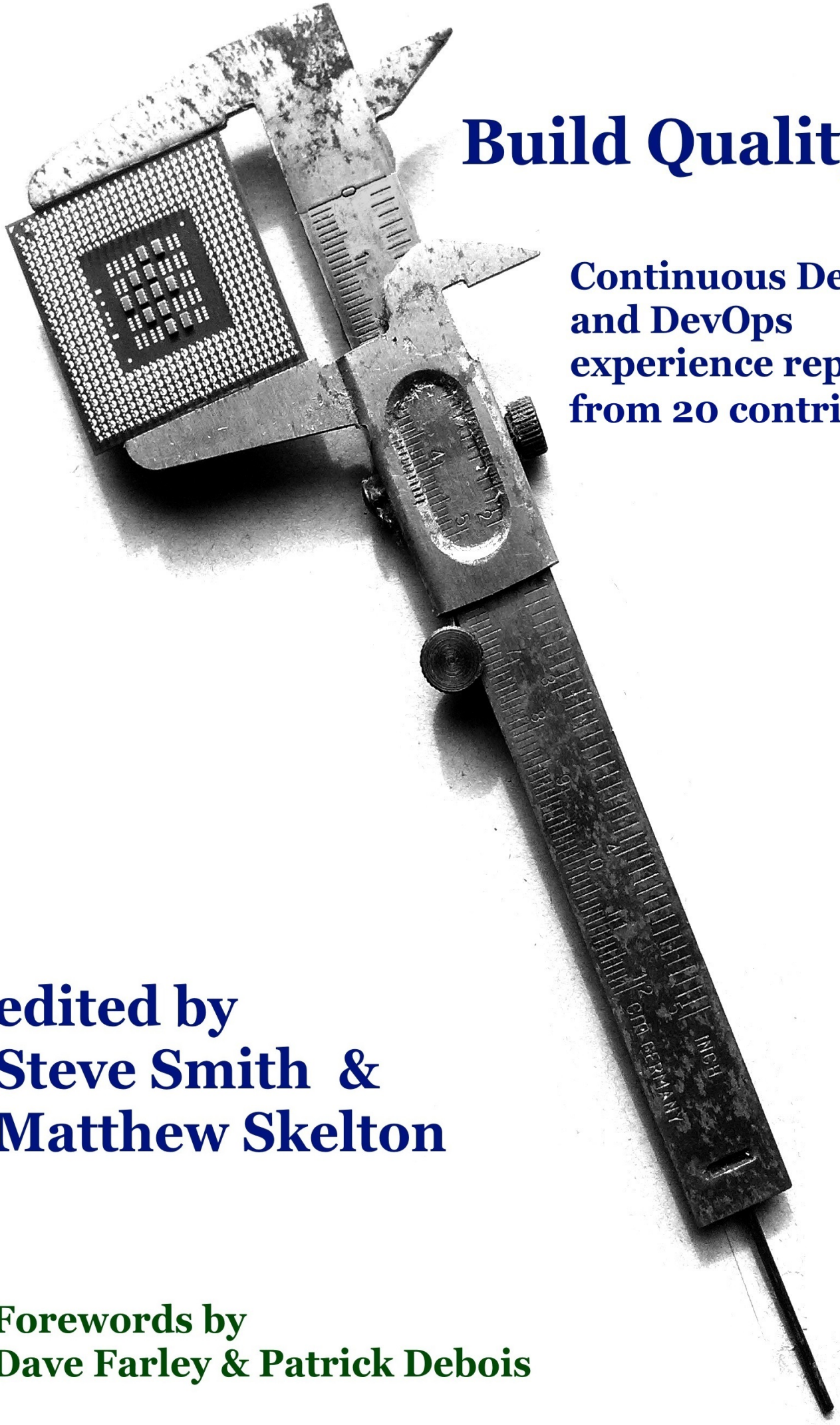
Build Quality In

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

Continuous Delivery Foreword - Dave Farley

Dave on Twitter: @davefarley77[1] Dave's blog: davefarley.net[2]

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

[1] https://www.twitter.com/davefarley77
[2] 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[3] - Patrick's blog: jedi.be[4]

Within the DevOps commununity there is a well-known acronym (first mentioned by John Willis) called CAMS[5]: '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.*

[3] https://twitter.com/patrickdebois
[4] http://www.jedi.be/blog/
[5] 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[6] - Chris' blog: blog.chrisodell.uk[7]

7digital[8]

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.

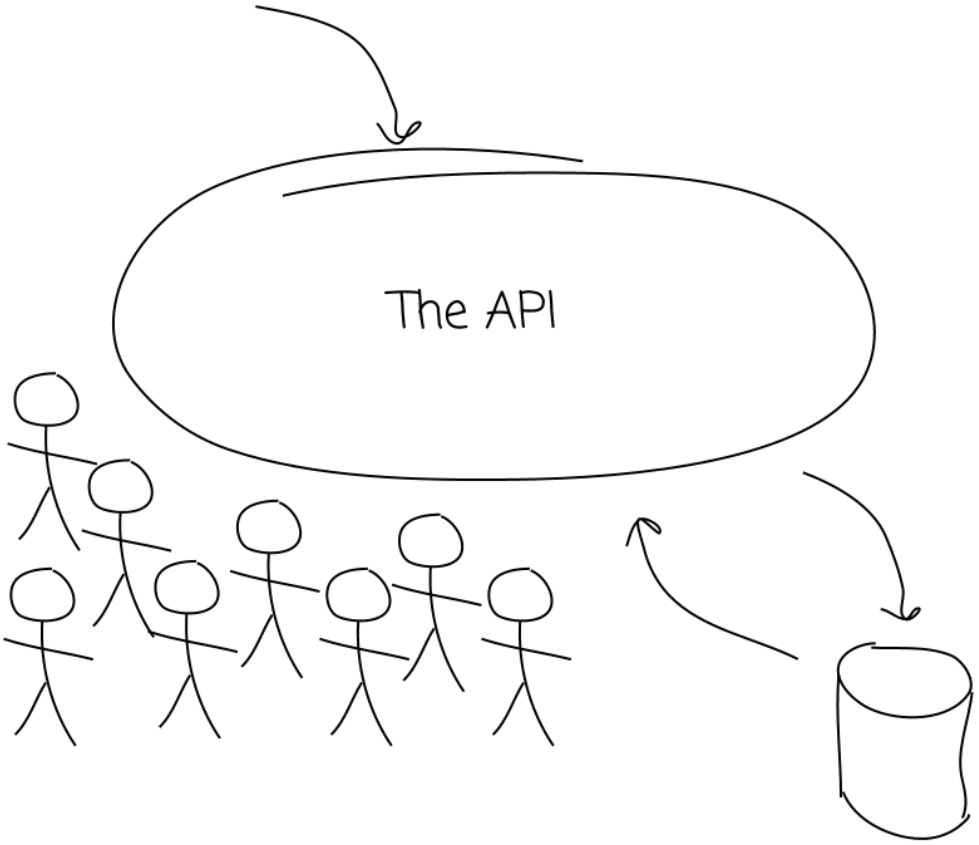
7digital also sold the collated music metadata of their catalogue to clients allowing them to build their own stores with the music files being supplied by 7digital. This data was provided in the format of large, ever increasing, CSV files. One client did not wish to receive the full CSV files, and asked 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.

[6] https://twitter.com/ChrisAnnODell
[7] http://blog.chrisodell.uk
[8] 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

[9]

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.

[9] 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.

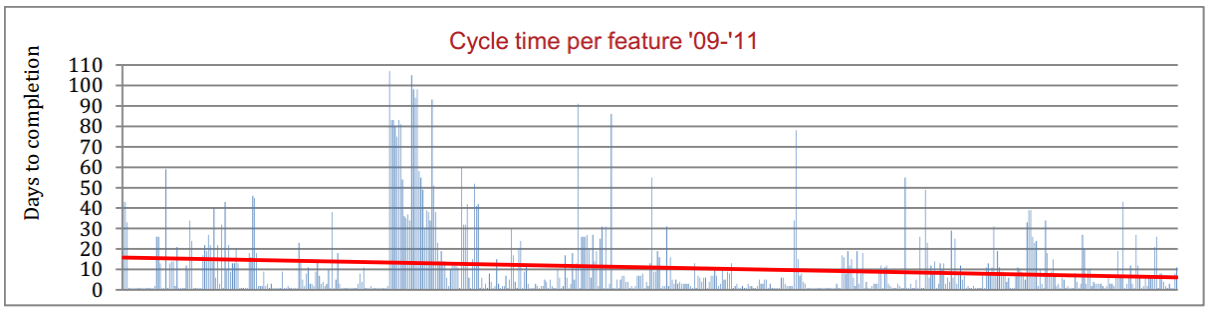
These changes significantly extended the time it took to fix a bug or add a feature and there were times when a refactoring was considered large enough to be tackled as its own work item. These items would be labelled as Technical Debt and prioritised in the backlog alongside the rest of the items. We 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[10], VP of Technology at 7digital, performed some analysis on this data which he published in a report in May 2012[11] and a subsequent report in 2013[12].

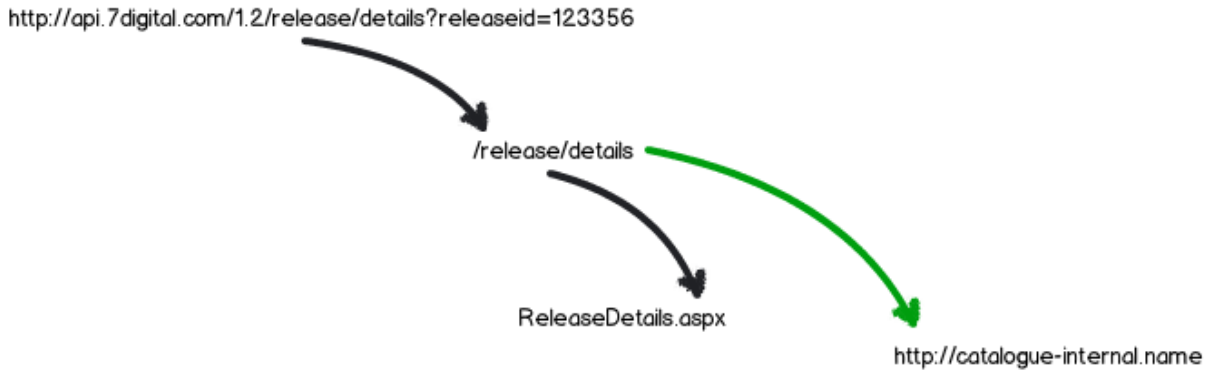
The interesting findings from the report show the team's cycle time during the period of refactoring greatly increased. The chart below shows a large spike where work items were taking more than 80 days to complete.



Cycle Time per feature 2009 to 2011

[13]

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

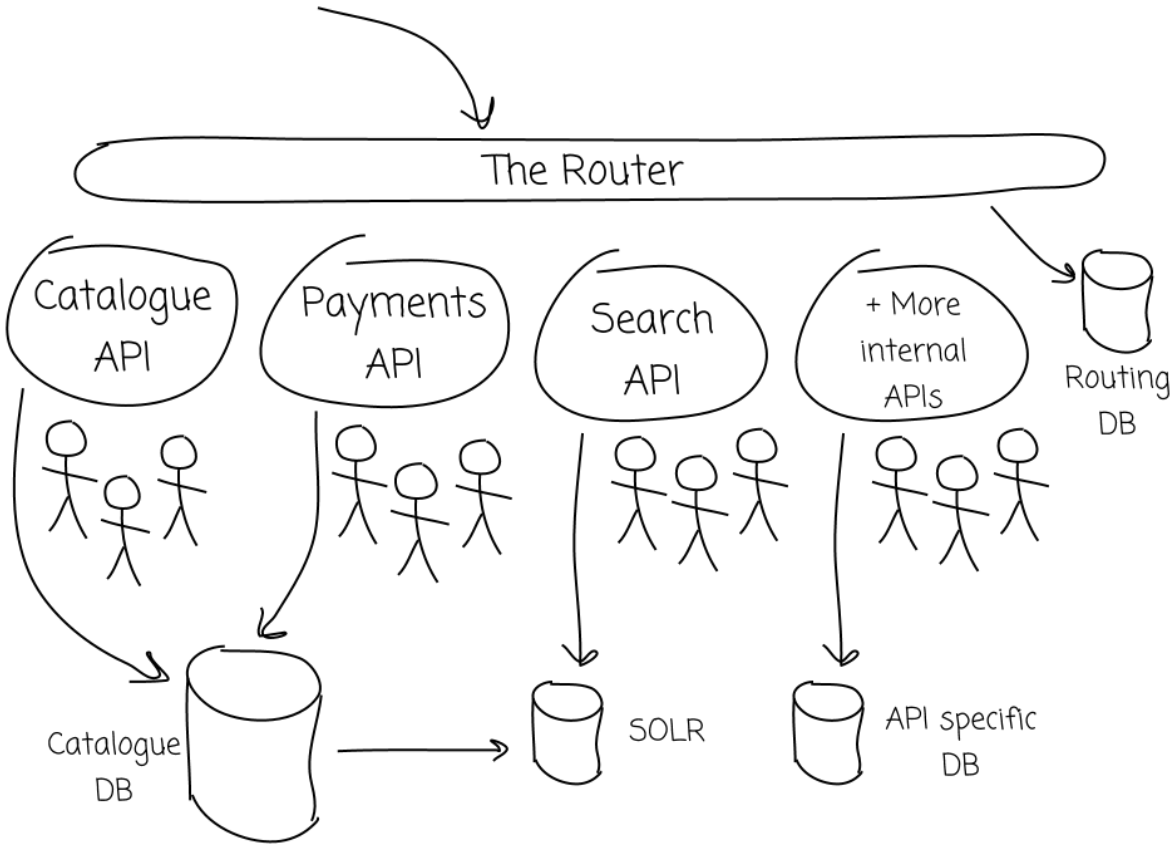
[14]

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.

[10] https://twitter.com/robbowley
[11] http://developer.7digital.com/blog/development-team-productivity-7digital
[12] http://developer.7digital.com/blog/7digital-development-team-productivity-report-2013
[13] http://developer.7digital.com/blog/development-team-productivity-7digital
[14] 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

[15]

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

[15] 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.

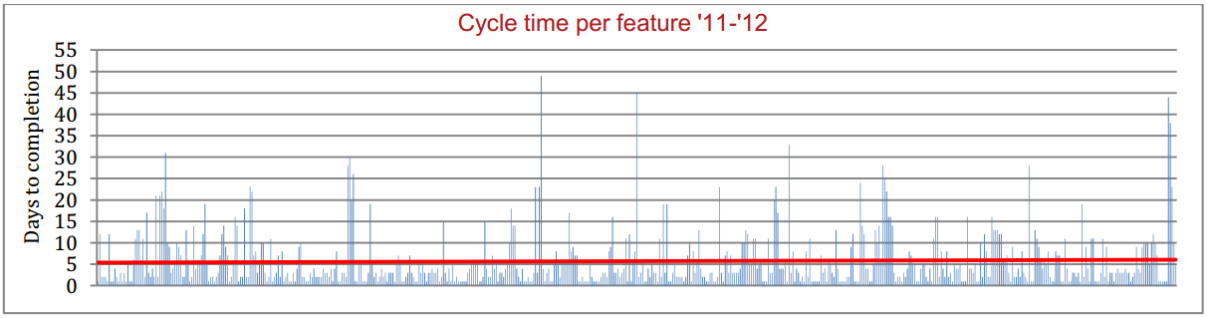
The frequent releases, rollback procedure and monitoring allowed us to spike out a change and test it in production with real live data. For example, if we believe that caching user details would be advantageous we could add a simple cache with a short timeout and monitor. If the spike proved successful 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.

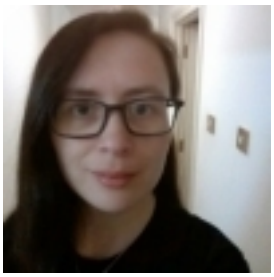


Cycle Time per feature 2011 to 2012

16

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

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

DevOps-ifying a traditional enterprise - Niek Bartholomeus

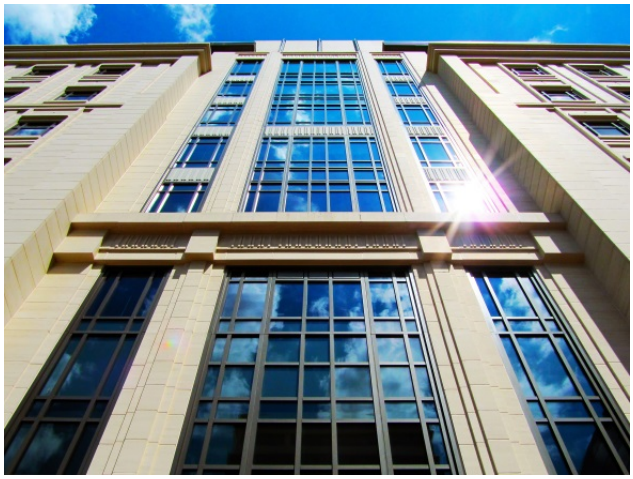
Niek on Twitter: @niekbartho[17] - Niek's blog: niek.bartholomeus.be[18]

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.



[19]

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

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

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

[19] 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.

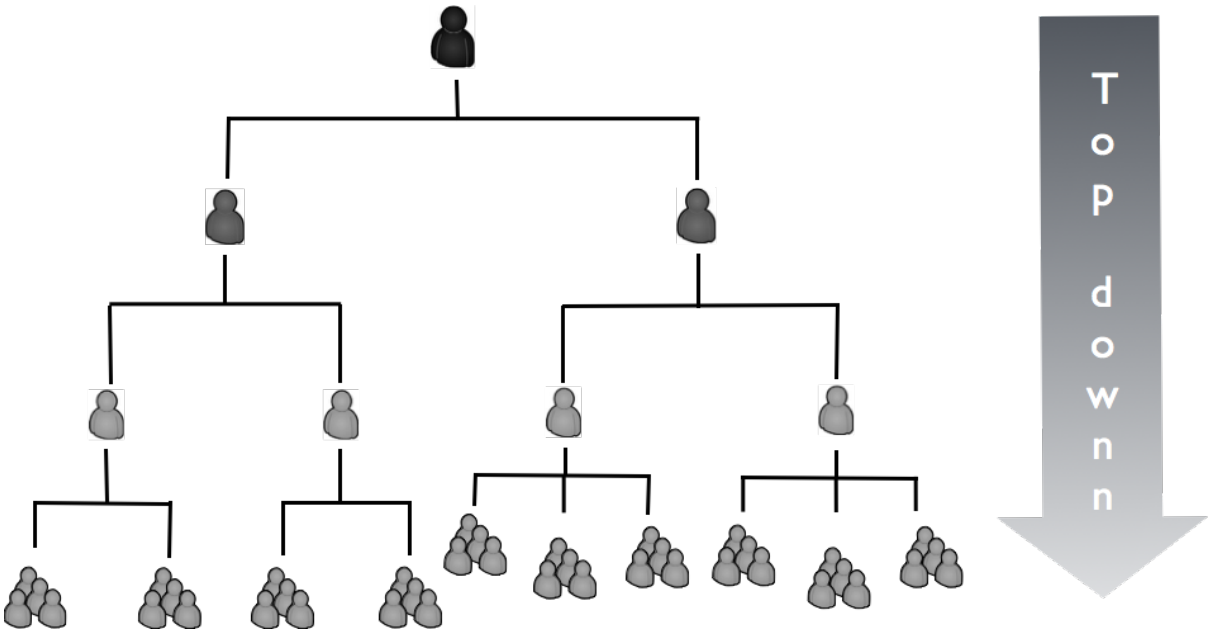
On the other hand, all that automation had not been able to contribute very much to the solution of what had by that time become the biggest bottleneck for delivering the software to the end users: the issue of the **infrequent, organisation-wide releases that remained very brittle and labour-intensive**. 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.

[20]

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.

[20] 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

 [21]

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.

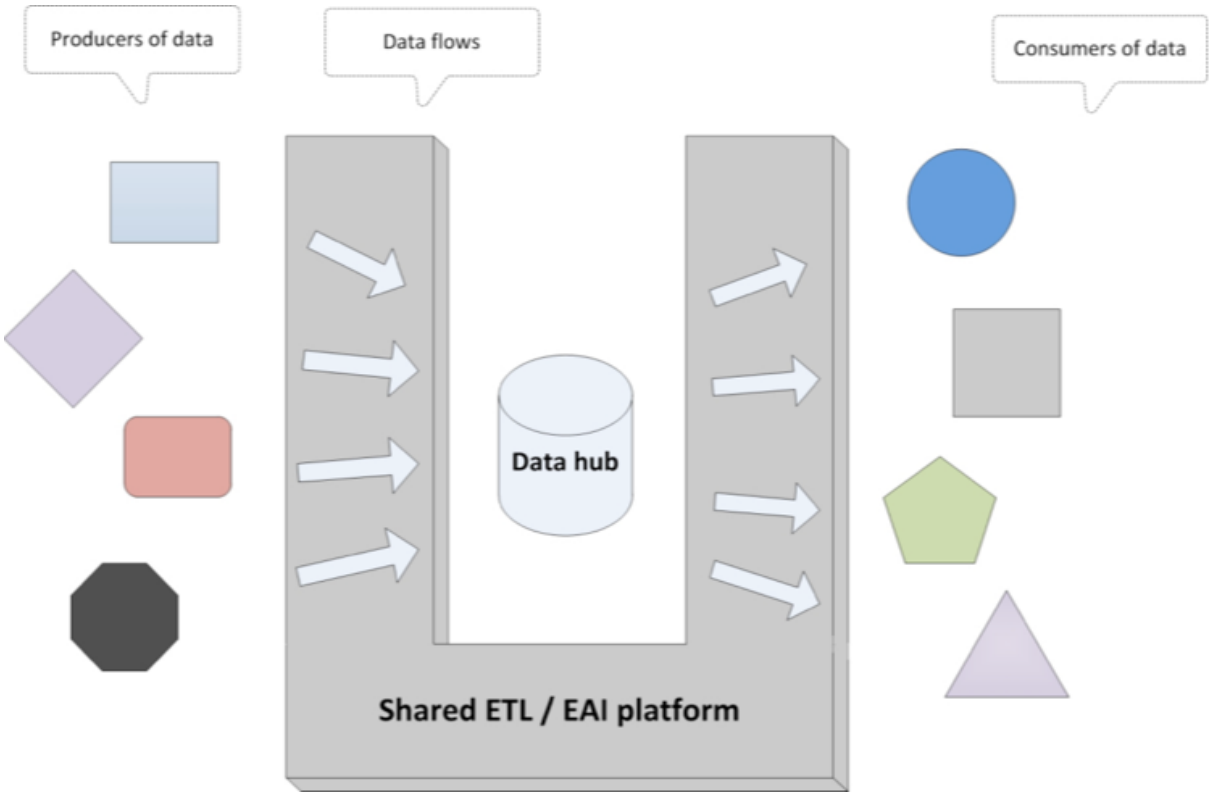
Another example concerns software delivery: as soon as the number of moving parts that has to be delivered into production reaches a certain threshold, an organisation-wide release management team is brought in to take control over the situation.

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.

[21] 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).

[22]

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

[22] 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.

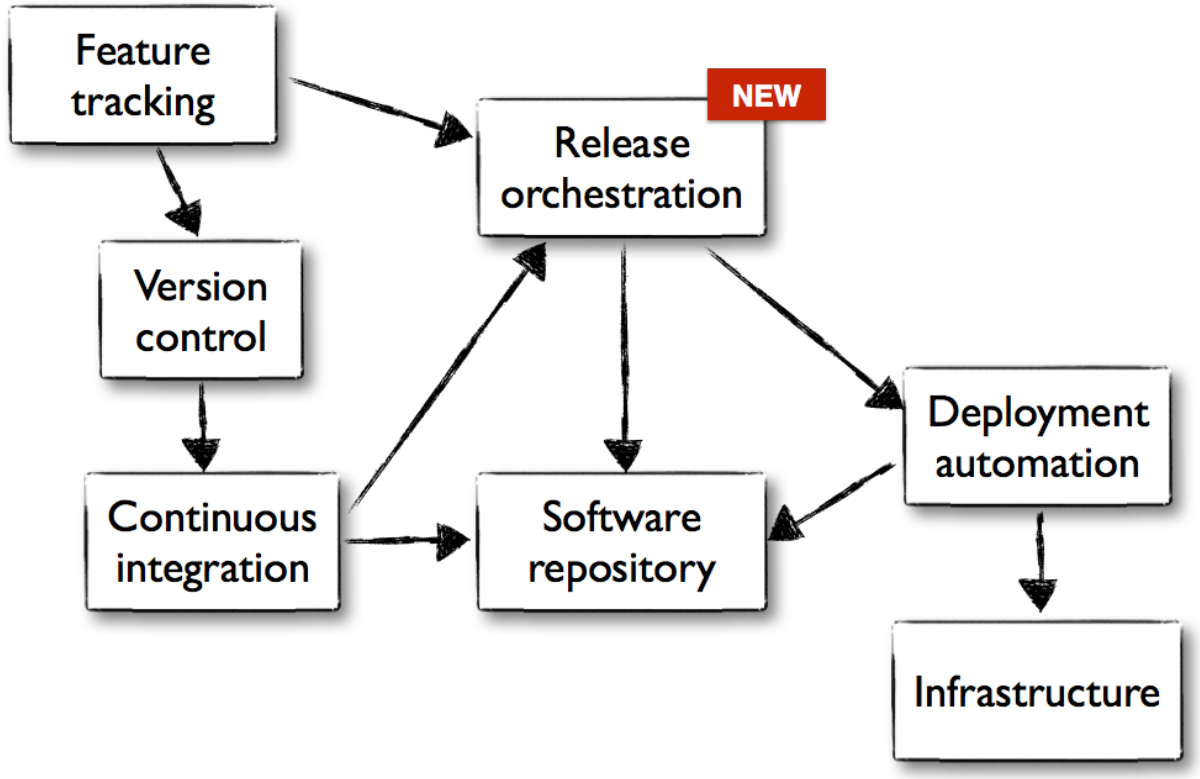
Tactical solution: enhancing the existing communication flows

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.

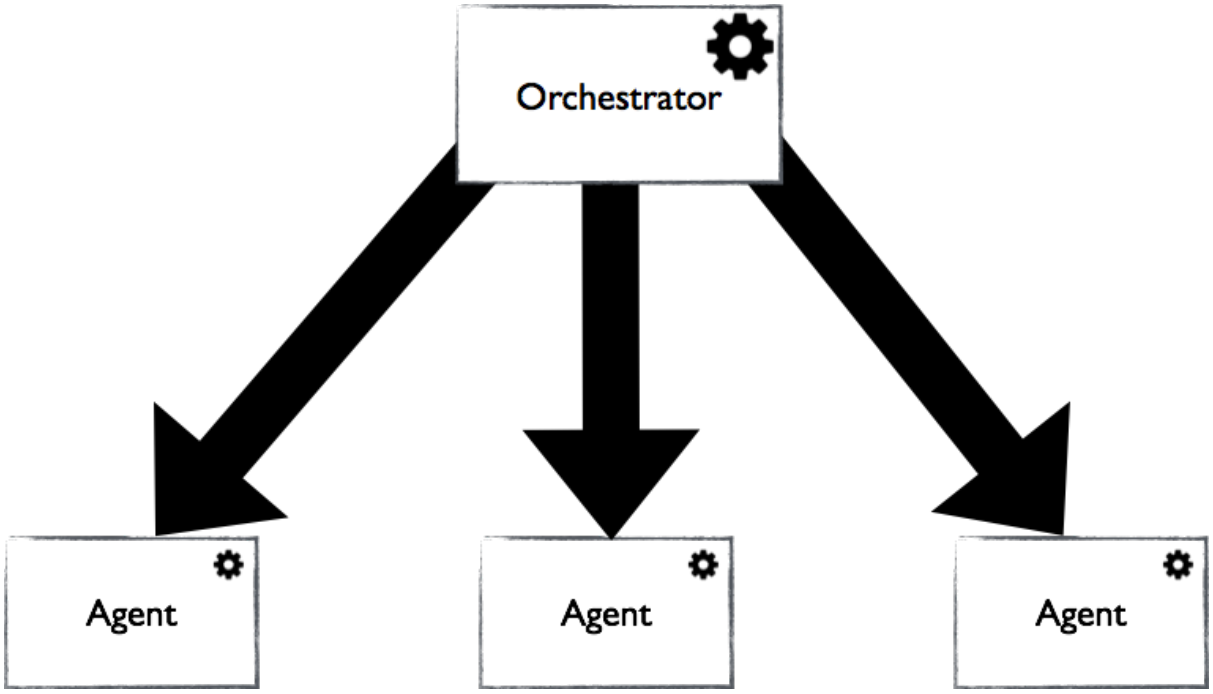
Structural solution: decentralisation

Scalability issues

The heavy reliance on centralisation that was traditionally used as a way to solve the data integration and release management problems [23] turned out to require a huge communication channel between the

[23] 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.

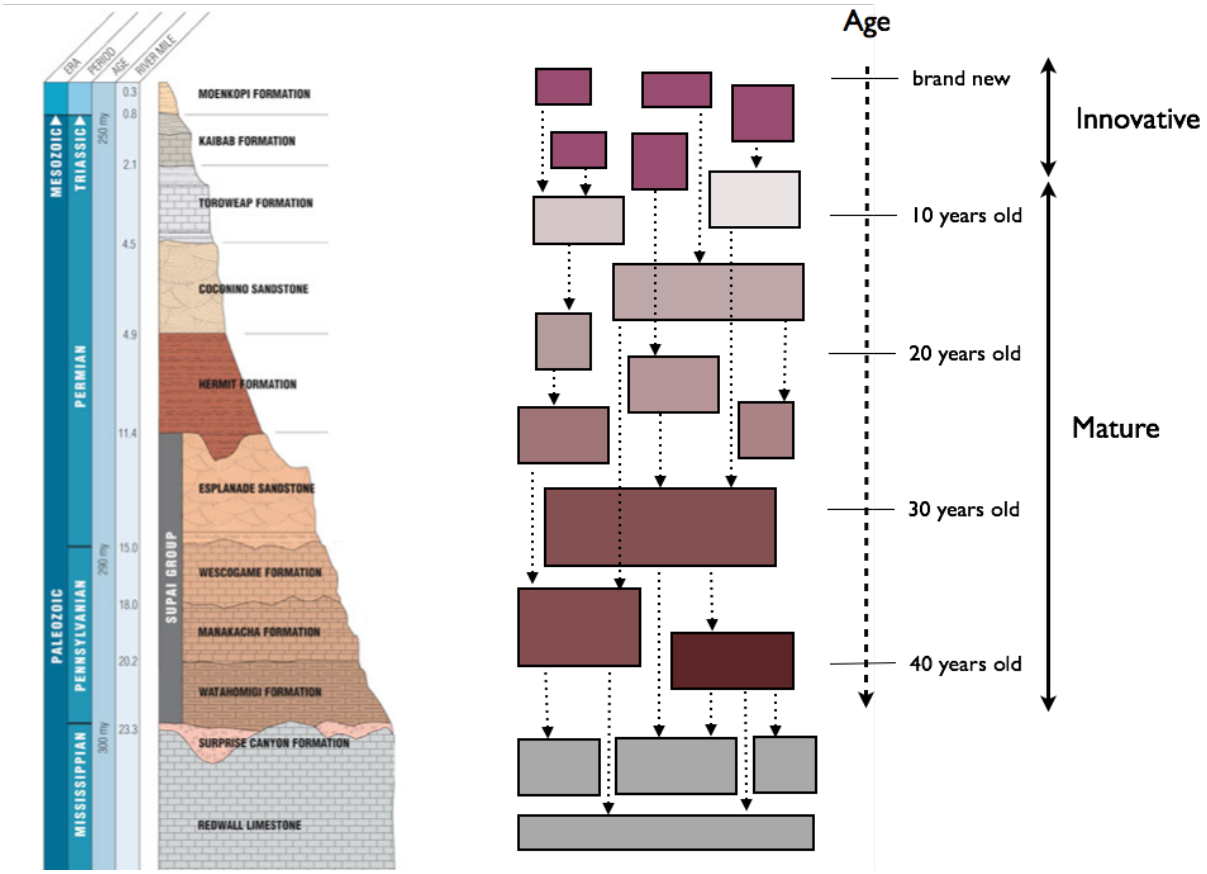


Centralisation of Orchestrator and Agents

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.

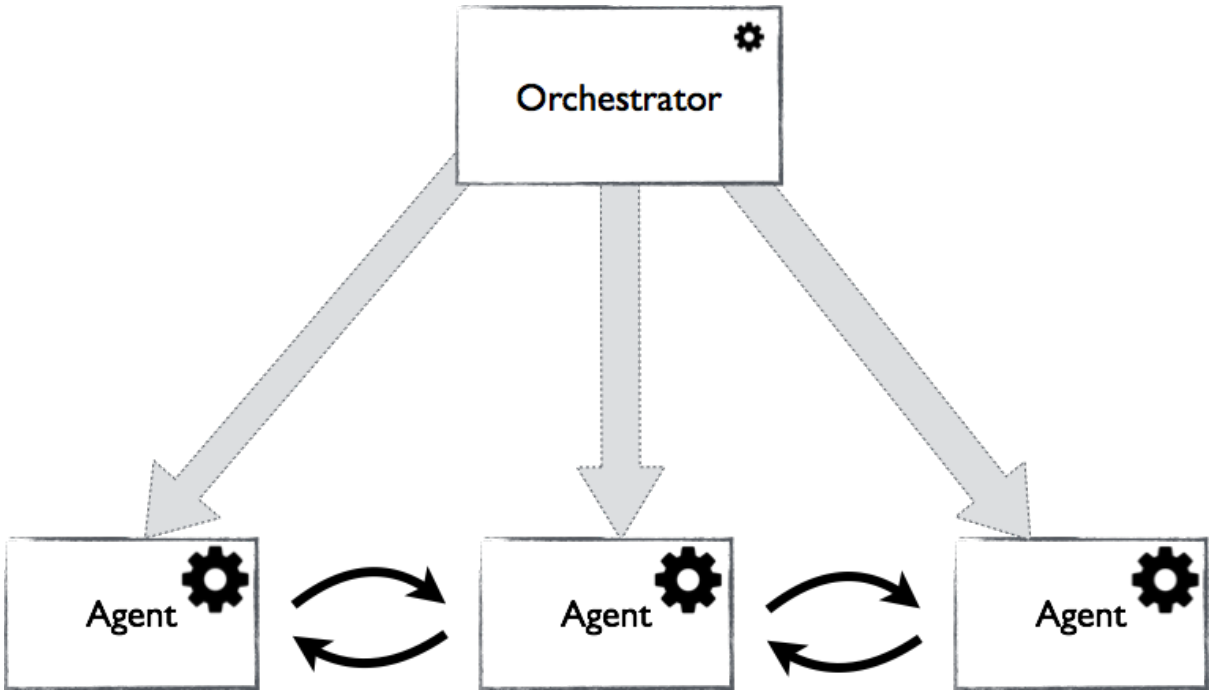


The 'palaeology' of innovative vs mature apps

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.


[24]

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.

[24] 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