FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

CONTINUOUS DELIVERY STORIES

eMag Issue 21 - January 2015

THIS ISSUE
Adopting Continuous Delivery

ARTICLE
Delivering Continuous Delivery Continuously

INTERVIEW
Dave Farley on Continuous Delivery

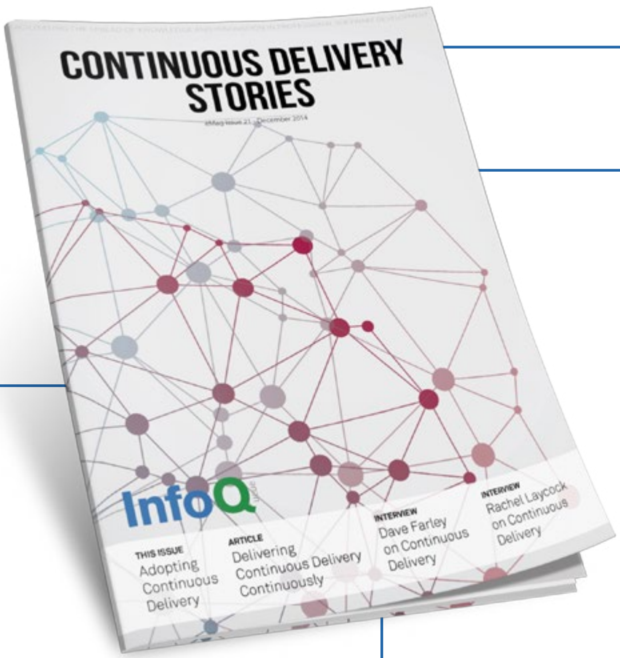
INTERVIEW
Rachel Laycock on Continuous Delivery

Adopting Continuous Delivery

Jez Humble addresses the most important factors in implementing continuous delivery: organizational, architectural and process.

Dave Farley on Continuous Delivery

Dave Farley discusses the reasons for Continuous Delivery and Continuous Deployment, the advantages and challenges they pose, and much more.


CONTINUOUS DELIVERY STORIES
InfoQ
THIS ISSUE
Adopting Continuous Delivery
ARTICLE
Delivering Continuous Delivery Continuously
INTERVIEW
Dave Farley on Continuous Delivery
INTERVIEW
Rachel Laycock on Continuous Delivery

Delivering Continuous Delivery Continuously

Simon Hildrew discusses the tools and processes used by The Guardian to create a continuous delivery pipeline.

Rachel Laycock on Continuous Delivery

Rachel Laycock explains her experience with bringing Continuous Delivery to companies, the main technical and social obstacles, and much more.

MANUEL PAIS is InfoQ's DevOps Lead Editor and an enthusiast of Continuous Delivery and Agile practices. Manuel Pais tweets @manupaisable

A LETTER FROM THE EDITOR

Continuous Delivery encompasses a set of strong practices to be successful and bring value to the organization implementing it, be it through reduced cycle time, more robust products, more visibility on current status, etc. But since continuous delivery can and should affect practices across the entire software lifecycle that means collaboration between different teams (Dev, QA, Ops, etc) is mandatory.

In short: reaping the benefits of continuous delivery is hard work! Culture, processes or technical barriers can challenge or even break such endeavors.

With this eMag we wanted to share stories from leading practitioners who've been there and report from the trenches. Their examples are both inspiring and eye opening to the challenges ahead.

Both Continuous Delivery book authors share their experiences trying to help customers effectively adopt continuous delivery.

Jez Humble tells us the most important challenges in implementing continuous delivery - organisational transformation, architecting for continuous delivery and effective collaboration - with case studies from HP, Amazon, Nokia and others.

In his interview Dave Farley delves into multiple aspects of continuous delivery such as differences between continuous delivery and continuous deployment; preconditions to continuous delivery; maturity models and anti-patterns.

Then we have Simon Hildrew, infrastructure developer at The Guardian, speaking about the media site's continuous delivery initiative. How it started, the approach followed (Deployment as a Service), the cultural challenges, the technology they developed to make it work and the resulting changes.

Finally, in her interview Rachel Laycock highlights the importance of architecting systems for release and breaking from dependency hell for any successful continuous delivery initiative to take place. At the same time she reminds us that organizations are politically driven and often resistance to change can't be broken with technological changes alone. ■

JEZ HUMBLE is a vice president at Chef, a lecturer at UC Berkeley, and co-author of the Jolt Award winning Continuous Delivery, published in Martin Fowler's Signature Series (Addison Wesley, 2010), and the forthcomingLean Enterprise, in Eric Ries' Lean series. He has worked as a software developer, product manager, consultant and trainer across a wide variety of domains and technologies.

Adopting Continuous Delivery

Jez Humble spoke at QCon San Francisco 2012 about the most important factors in implementing continuous delivery - organizational, architectural, and process - with case studies from HP, Amazon.com, Nokia and others. This is a summarized transcript of his presentation.

Since the Continuous Delivery book came out in 2010, Dave Farley and I have been working with a bunch of different companies, helping them to adopt the ideas that we talk about in the book. Sometimes, it's gone really well. Sometimes, it's gone okay. Sometimes, it's been a total disaster.

Continuous delivery is hard because of organization, architecture, and process: people are terrible at these but not because they lack the tools.

You need to know why you want to change your organization. What are you going to do? What's the measurable change that you're going to achieve? Do you want to reduce lead time? Do you want to increase your number of releases? Do you want to reduce the amount of wasteful development?

You also need acceptance criteria that you can measure for the changes you want to make to your organization. You want some way to achieve that change rapidly. You should never have a planning horizon of more than three months.

The biggest problems people face are organizational structure and organizational culture, and then the architecture of their systems. Conway's Law basically tells you that your organizational structure and the architecture of your systems are tightly coupled. And the fact that they're coupled makes them really hard to change.

What I'm trying to say is: don't start a big-bang continuous-delivery project. First, work out what your biggest constraint is. What's stopping you from going to production earlier and fixing that problem?

Why continuous delivery?

What is continuous delivery? Continuous delivery focuses on optimizing your process for delivering incremental changes to your customers. You want to make it cheaper to put out incremental change in terms of cost, time, and risk.

The Standish report that came out in 2002 said that more than 50% of the features that you build are either rarely used or never used, and that figure increased in the 2006 report. That's the biggest source of waste in your jobs: stuff that you build that no one uses.

One of the main goals of continuous delivery is to obtain fast feedback on hypotheses so that you avoid building ideas that no one cares about.

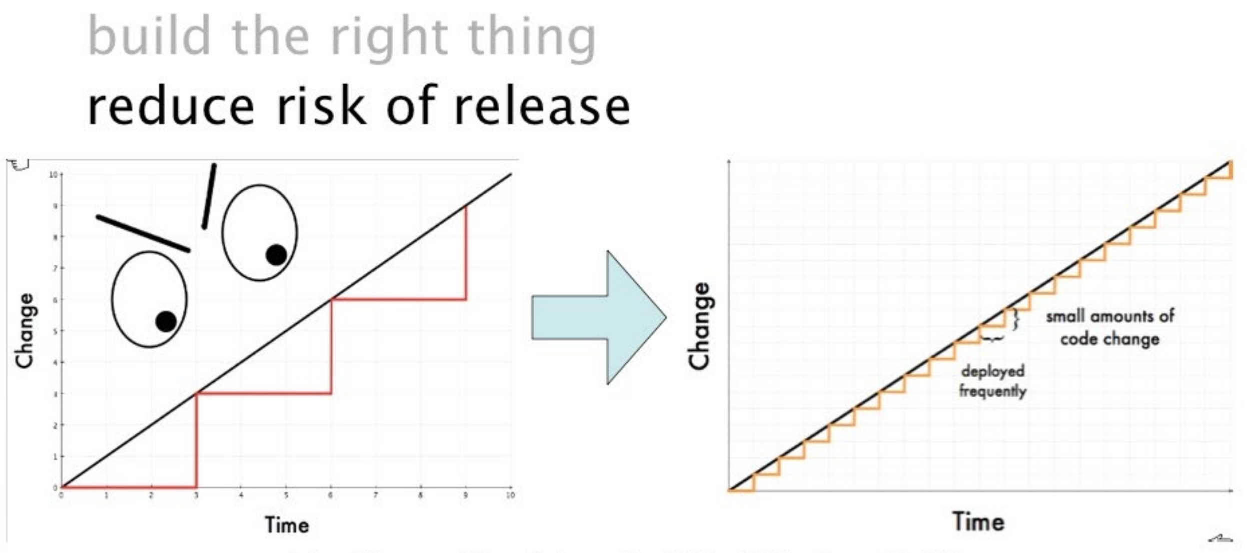
This requires a culture of continuous experimentation where, instead of starting from "Here are the requirements for the features we're going to build," we start from "Here's my hypothesis, my guess about what may be valuable to my customers, and I'm going to do the minimum possible amount of work to get data from my customers to determine whether I should invest real effort in building this thing."

You want to optimize your process to get around this loop as fast as you possibly can.

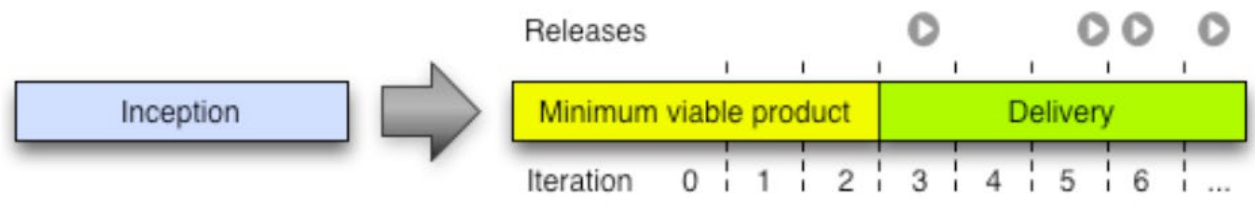
The second reason to adopt continuous delivery is that it reduces the risk of release. If you release every three months, that's three months of stuff that can go wrong. It's harder to find and fix problems.

The third reason is that stuff that's live in production is the only real measure of done. For a developer, done means "it works on my machine" but done-done means working at scale with realistic datasets in production.

Most of the pain and risk in software projects is between dev complete and live, where



John Allspaw: "Ops Metametrics" http://slidesha.re/dsSZlr



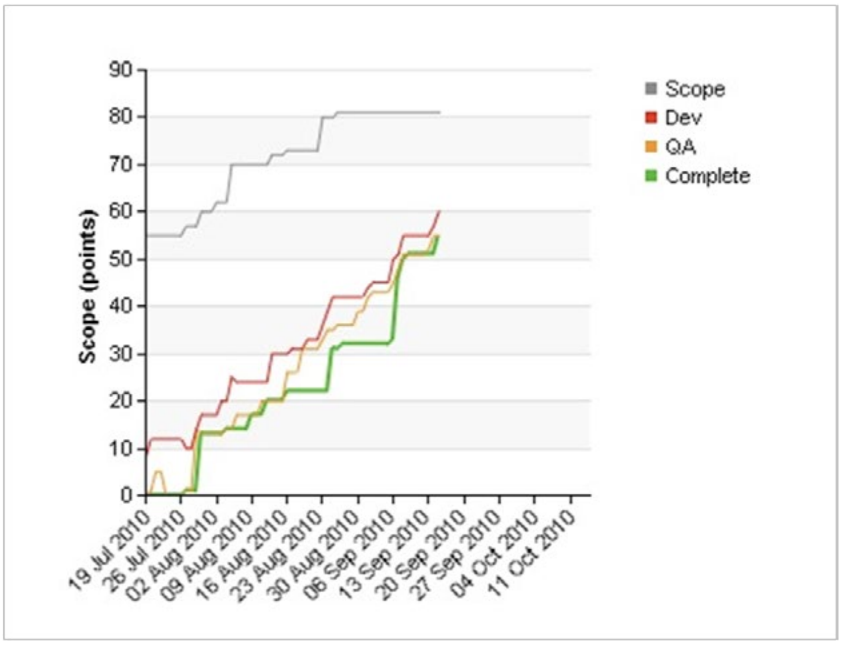
you discover your architecture doesn't support the number of people that you want to run in production.

So it's really important to have a burn-up chart, a cumulative flow diagram. From the diagram below, you can see two important things at a glance. The Y-axis is scope. The X-axis is time. The gray line is the amount of scope in this project.

There's a red line to indicate the amount of development work and an important green line that means code live in production: not QA-passed, not customer signed-off, but deployed. It's not necessarily live in the sense of everyone can see it, but it's at least deployed to production.

You can see from this graph two things at a glance. The vertical distance between the red line and the green line is batch size, the work in progress. The horizontal distance between the red line and the green line is lead time, time from check-in to release.

You can see at a glance that the red and green lines are closely correlated. The cheapest way to reduce lead time, which is what we're trying to optimize our process for, is reduce batch size, to reduce work in progress. That is exactly the value proposition of continuous delivery.



How do I know if I'm doing continuous delivery?

For a start, any time you want to put out a new product idea or any new product, you do an inception.

The purpose of the inception is to work out "What's the thing we're going to build? What's the vision of what we're going to build and how will I get real customer data as soon as possible on whether my product's hypothesis is actually going to be valuable to my customers?"

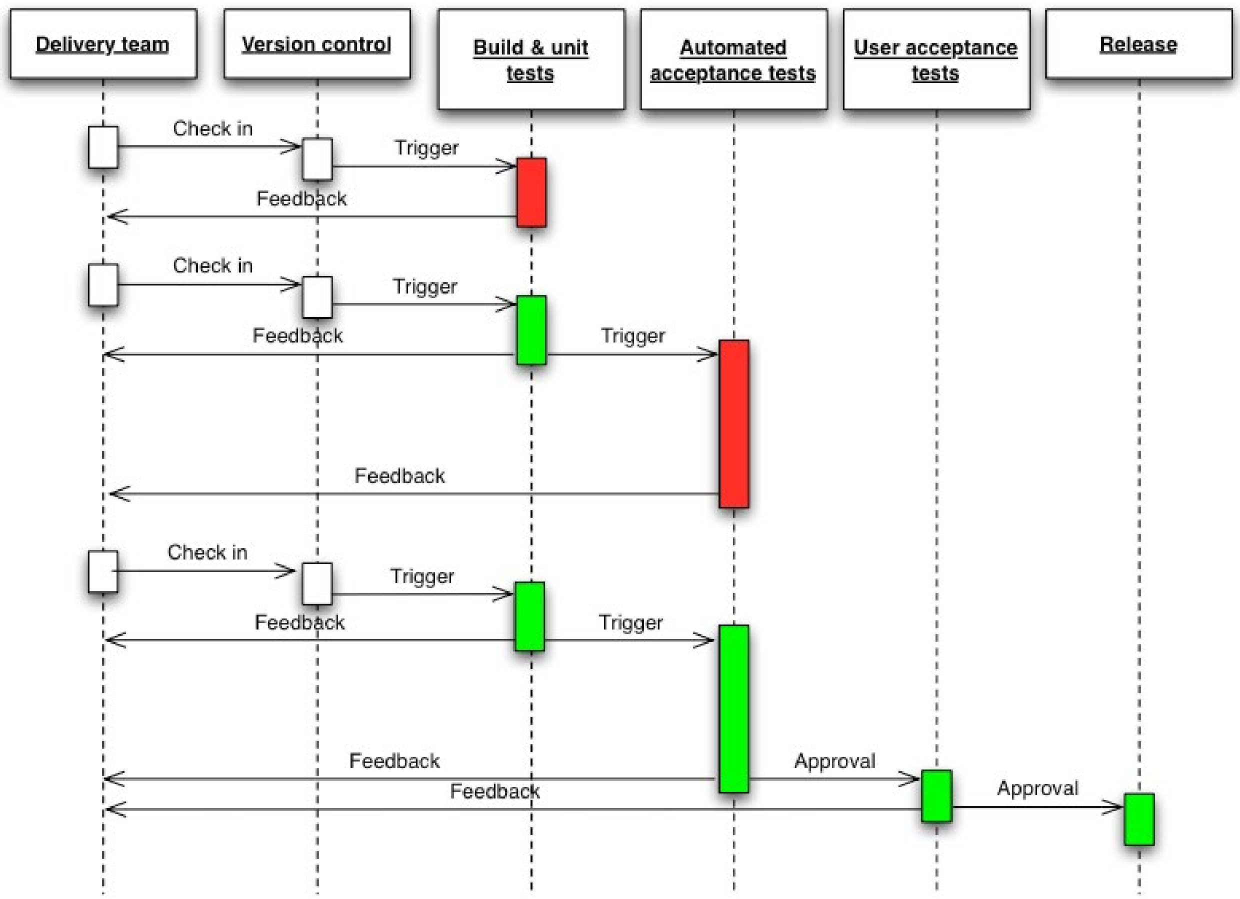
Get customer data as soon as possible and then work out whether to keep pushing out new increments or to pivot and try to achieve the vision with a different strategy or to just abandon the idea and not waste any more money.

When you're doing continuous delivery, your software should always be releasable on demand right from the very first feature you build, which hopefully will be something like a status page that tells you what version you came from. So right from the very first feature, we make sure that our software is always releasable.

Then, crucially, you prioritize keeping the system releasable over writing new features. If at any point the system is not deployable, you stop the line. You don't do any more work. You focus on fixing the tests, making sure the deployment scripts are working, and fixing anything that is stopping you from pushing trunk live. That takes priority over any new work.

Finally, you want to make sure that people know as soon as possible if they've broken things. So you want to prioritize fast feedback to people about the effect of their change, whether that's a change to the source code, to the infrastructure, to the database schema, or to the configuration of the system.

The system for achieving this is called the "deployment pipeline". The idea is that when you make a change to version control, your continuous-integration (CI) system runs the build and unit tests, and in a few minutes supplies feedback as to ▶



whether you've done anything stupid.

If the build or the tests fail, everyone stops what they're doing. No one is allowed to check into version control. You focus on fixing that problem.

For anything that passes the build and unit tests, you run a more comprehensive set of automated acceptance tests, which might take longer. If that fails, you stop the line again and focus on fixing that. People can still check into version control but someone on the team has fixing those automated tests as their highest priority.

Any build that passes the acceptance tests goes downstream, potentially to usability testing or performance testing, and then, ultimately, to production.

In continuous delivery, every change results in the build and every build is a release candidate. The job of the deployment pipeline is to prove that this release candidate cannot be released. If you can't prove it then you should feel absolutely fine pressing the buttons to release it. If you release your build and something breaks anyway, it means your validations aren't good enough.

The Greenfield project

A project that I want to talk about is a project run in the UK with a small government team. The product owner was a contractor and they hired ThoughtWorks to help them build it. We could do whatever we wanted. We said, "We'd like to use continuous delivery."

They said, " We don't know what that means, but sure, go ahead," which is really great.

First of all, we worked out what the plan for the product would be and we included all the tasks for continuous delivery like implementing the deployment pipeline composed of the test and deployment automation, and setting up the infrastructure in a fully automated way with Puppet.

We said, "This is going to be about 25% of the work."

They were like, "We don't really know what that means, but sure, go ahead and do that," which was awesome.

We ended up with four pairs of developers. We had one pair dedicated to putting in place the infrastructure automation and the deployment pipeline, all that stuff. We also made sure that the system was architected for continuous delivery from the beginning.

One of the first things that the team did was work out how to automate provisioning of production infrastructure and test infrastructure using Puppet. They chose Ruby on Rails as the stack, and one of the reasons they did that is that session state in Rails is in the database. If you want to do zero-downtime deployments, you can and not worry about session state w which is one of the problems people have with Java stacks if they don't handle session state in a sensible way.

We were dependent on a content management system (CMS) that was not able to do zero-downtime deployments even though it was clustered so we put some architecture in place to cache that data while we were doing an upgrade. That way, you could still get the data as read-only from the CMS even during CMS upgrades.

So we thought about all these architectural constraints, and planned for continuous deployment and continuous testing as part of the architectural development of the system. That was all built in.

The biggest problem was the skill. We needed to understand how to do Puppet, how to configure infrastructure, how to do automated testing. Those skills are still not widespread in our industry. That was our biggest constraint.

The good thing was the project launched early so we deployed to production well before the release date. On the release date, all we had to do was turn it on and put out the e-mail – send out the marketing literature so people would come to the site that had been deployed for a couple of weeks.

We were able to deploy on demand multiple releases per day but the business didn't want that. It was problematic because a release required coordination among the support people and the marketing people. When the business comes to you and says, "You're deploying too often. Please slow down with that," – ethat's when you know you're doing something right. The constraint was the ability of the business to come out with new hypotheses, not the ability of the development team to build and ship them.

The path of continuous delivery

A second story I want to talk about is the HP LaserJet firmware team. These are the people who build the firmware and the operating system that HP LaserJet printers run on. In 2008, they had a problem: they were moving really, really slowly.

Some senior people in the engineering organization looked at what the team was doing. What they found was really unpleasant. Ten percent of the team. time was spent on code integration and 20% was spent on detailed planning.

The team had a separate branch in version control for every model of printer that HP built, so they spent another 25% of their time porting code between branches. Additionally, 25% of their time was spent on current product support and 15% on manual testing.

When you add up all that time, what you have left is 5% supposedly being spent on product innovation. If that were me, I would have spent that 5% of time sitting down with lots of Aspirin in my body.

They had a serious problem, and what they did was quite radical. They re-architected their entire system from the ground up to allow them to check in on trunk. Instead of having a different branch for every model of printer, the firmware and the OS would start, would look at which printer it was running on, and would turn features on and off using feature flags.

The team then put a large amount of automated testing in place. Not only do they implement unit tests, but they also added 30,000 automated functional tests that run in a virtual environment and then on logic boards. ▶

IN CONTINUOUS DELIVERY, EVERY CHANGE RESULTS IN THE BUILD AND EVERY BUILD IS A RELEASE CANDIDATE.

They now have continuous integration in place so anytime someone checks into version control, the system builds and runs the tests. Anytime someone checks something into version control and it doesn't pass the tests, the build system automatically reverts the change out of version control.

It took them three years to rebuild everything. They started with a new architecture. By the end, they had spent 2% of their time on continuous integration and 5% on planning. They had one branch for everything, so they no longer ported codes. The team estimated they were now spending 40% of the their time building new features.

If you want a return on investment for continuous delivery, here's the economics: development costs dropped by 40%; programs under development increased by 140%; development cost per program fell 78%; resources driving innovation increased by 5%.

The beginner error that people make about lean is that lean is about reducing costs. Lean is about reducing waste in the value stream. It's about improving return on investment. It's going to cost you more initially, but that's okay because – assuming your organization is going to survive more than two years rhere's the return on investment.

Organizational transformation

So, you want to change your organization. You want to run experiments to work out how to make your organization better and get measurable data as soon as possible on the effects of the change. Sometimes, people ask, "Should I change my organization top-down or bottom-up?" You need both.

There has to be executive sponsorship from people who understand that the change will not just be about cutting costs. It's going to be about decreasing waste in the value stream. You need people who are enthusiastic about this. People need to care and people need to want to get better. Once you've got that, everything pretty much takes care of itself.

A really good place to start is with continuous integration. Continuous integration tells you if you broke something. The rule is if you break it, you fix it. You're creating a system in which people are made aware of the consequences of their actions and they are forced to take responsibility for those actions – that's what CI is.

Continuous integration is a practice. It's not a tool. You don't need a tool. When you scale, you need a tool. But running Jenkins on your feature branches is not continuous integration. That's the crucial thing that I want to emphasize. Continuous integration is the practice of keeping the system working and making sure that this your team's highest priority.

Architecture

You need to architect for continuous delivery as well. The biggest reason that organizations can't adopt continuous integration is often because they have a "big ball of mud" architecture. Any change affects everything and conflicts with everyone else. So, everyone wants to work on feature branches so that they can develop crappy, undeployable code much faster without having to talk to the other developers.

CI means you have to check in, which means you have to talk to other people, and that's unpleasant for developers. Version control is a communication mechanism. That's the primary purpose of version control. When you force people to use it, it starts to be awkward.

If people have to feature branch, that's a sign that there's something wrong with your architecture. A colleague of mine named Dan Bodart has this really nice quote. "Feature branching is a poor man's modular architecture." It means you don't have a modular architecture.

The solution to architectural problems is service-oriented architecture.

It's notoriously difficult to get any information out of Amazon.com but this is a story worth telling. In 2001, Amazon had a big ball of mud. Jeff Bezos, the CEO of Amazon, sent a memo to the entire technical staff to say that their highest priority was to create a service-oriented architecture.

He hired a former Marine to visit all the teams and make sure they were only talking to other teams via the API. If you tried to access another team's data via the database, the Marine would shout at you and if you didn't respond meekly, you would lose your job.

Most organizations don't treat their architecture with this level of seriousness. Instead, they create an architecture group into which the people who have been in the organization a long time may retire and issue edicts in UML that everyone ignores. Not at Amazon.

It turns out that Jeff Bezos is a really smart guy. What he was doing was forcing the organization to build a platform, which was one of the major reasons why Amazon is so successful today.

AWS was developed by a rogue group of Amazon employees based in South Africa. The point of AWS was not to create an infrastructure for Amazon. It was to create a platform that people could build on. Eventually, it got absorbed into Amazon, and the company now runs all its stuff on AWS.

If you need to make large-scale architectural changes to your system, there's a pattern called "branch by abstraction" to use, and Amazon was able to do that. If you have a monolithic legacy system that you want to replace, don't build a new system to replace the old one that you'll roll out all in one big-bang go at the end of two years. That always goes wrong.

You want to pull off small parts of your system and have the rest of it talk to those new components that you're writing. Piece by piece, you pull the functionality of the old system and replace it with new parts. Eventually, all that's left is a new system.

Design for test and deploy

You need to design for testable and deployable systems. The time it takes you to restore service to your customers should be deterministic. You want to be able to predict mean time to restore service.

In order to do that, you need to have a completely automated process for provisioning and managing and for making changes to your infrastructure so that if your system goes down in some disaster, you can restore service in a known time using an automated provisioning process.

It should be possible for developers to set up an entire integrated system on their dev boxes using virtualization. If your system requires multiple hosts in order to run, you need to be able to set that up on a dev environment using virtualization, using a push-button provisioning process.

It should be possible to run acceptance tests in a non-integrated environment and have a high level of confidence that when you run this component in an integrated environment, it's going to work. You need to create test doubles for the external systems that your component depends on in order to be able to run those acceptance tests end to end.

Director of continuous delivery

This next story is about an organization everyone's heard of, but I can't name it. The company wanted to do continuous deployment. They had a Web front end that was built by one part of the organization and a mainframe-based booking system that was developed by another part of the organization.

The Web front-end people hired us to do continuous delivery, but we couldn't get there essentially because the mainframe people were working waterfall. Anytime we made a change to the Web system that required a change to the mainframe, we would have to wait until they could deliver it. Because of that, we couldn't ever get to continuous delivery.

But, in a year, they went from one release a month to two releases a month. They reduced cycle time by 40% and they focused on the low-hanging fruit. They reduced the number of feature branches. They never managed to completely eliminate them but they did eliminate the most volatile branches. They got from a hundred branches down to about a dozen, and those branches did not change very often. They didn't completely fix it, but it made a big impact.

The other thing they did was program-level coordination. It turned out that there was no regular meeting where the Web people could compare what they wanted to do with what the mainframe people were planning. Just having regular meetings where the two groups talked to each other about what they were planning made a huge difference to their ability to deliver on their plan.

The other thing they did was to hire a director of continuous delivery whose job was to manage the initiative. This person didn't have any power but would help all the teams get better at what they were doing. That kind of centralized resource to provide coaching and training and encouragement is often important in helping these things bed down.

The Nokia test

When you're trying to change a whole organization, you want to do it team by team. You need to start with finding the right team, a team that's going to be excited about doing some of this stuff. They need to have some level of capability, too.

Bas Vodde was hired by Nokia Siemens to help them do Scrum, and he was trying to find the mature teams. So he came up with the "Nokia test" to find out which teams were doing something even remotely Scrum-like.

He would informally ask the team a few questions. Are your iterations time-boxed to less than four weeks? Are your software features tested and working at the end of every iteration? Does the iteration start before a specification is complete?

Are you doing Scrum? Do you know who the product owner is? Is the product backlog prioritized by business value? Does the product backlog have estimates created by the team? Are there project managers or others disrupting the work of the team?

Try to find the people who have the best chance of ►

succeeding because they have some decent level of capability, and start with them. From helping them implement this stuff, you will learn lessons that you can take to other teams as you gradually, slowly, work your way through the organization until you get to the people who are the laggards.

How not to do continuous delivery

The final story I have concerns an organization that utterly failed to implement continuous delivery in any way. They hired us because they wanted continuous delivery – but what they actually wanted was for us to wave the magic continuous-delivery wand and for the magic continuous-delivery fairies to rock up and change all their stuff without them having to change or do anything.

Obviously, that doesn't work. The biggest problem was the company's monolithic architecture. They had around 70 million lines of codes and they had a tool for adding new functionality called the Vertical Slice Generator.

To use the Vertical Slice Generator, you would first write the SQL query for the new function. Alarm bells are already going off, hopefully. Then the Vertical Slice Generator would automatically generate the stubs and classes to allow the UI and the business logic for the SQL query to be created. Then the developers just make that stuff work. The business logic, by the way, lived in stored procedures so it was untestable. There was lots of duplication, which the architects called "unplanned reuse".

The company didn't want to change the architecture. They were using water-Scrum-fall. It was untestable. It was undeployable. They wanted us to bring in the continuous-delivery fairies. It's not possible.

My favorite comment from the person who tried to help them implement this stuff was this: "They don't need a deployment pipeline. They need to talk to each other much more."

People are the key

All problems ultimately are people problems. You need to create a culture in which people talk to each other, they collaborate effectively, and they're getting automatic feedback on their actions so they can learn how to do things better.

At the beginning of every new endeavor, get everyone together, including the support people, the ops people, the testers, the devs, the prototypers, the business people, etc. And keep meeting.

LCD screens should be everywhere. All those nice graphs and stuff you have in ops should be available to everyone. Everyone should have access to all the information they need, and they should be told when something they've done has caused something to go wrong so they can fix it. Create a system in which people are made responsible for the consequences of their actions, and they will learn.

You need to understand why you want to change. You want to get some kind of measurable change as fast as possible even if it is going to take years to get to your goal. Start with CI. Organization structure and architecture are normally the two places with problems. You want to create a learning organization where people can learn because they can see the effects of what they're doing. ■

READ ONLINE ON InfoQ

Dave Farley on Continuous Delivery

Dave Farley is co-author of the Jolt Award winning book Continuous Delivery. 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.

Interview with **Dave Farley** by **Peter Bell**

Peter Bell spoke with Dave Farley at the 2013 QCon New York. Here is an edited version of that conversation.

I wanted to ask you some questions about continuous delivery, the practice you work on, and of course the book that you wrote with Jez Humble. For anyone who is new to continuous delivery, how would you describe it, what is it about?

I think there are several different ways of looking at it. Perhaps the most straightforward for most people that are familiar with continuous integration, is it is kind of like the extension of continuous integration across the software-development life cycle.

I am a popular-science nerd and so I'm very keen on trying to apply that kind of rational thinking to my day job of writing software so for me continuous delivery is trying to use the scientific method, trying to achieve verifiability, and trying to apply the skeptical mind to the work that I do. So I don't want to make assumptions, I don't want to make guesses about the way in which my software is going to be robust, fulfill its functional needs, and be deployed into production. I want to be able to assert those things before I release them. And so for me continuous delivery is trying to do that, trying to automate much of the development process as seems useful and leave human beings do the creative bits. ▶

THERE'S A FUNDAMENTAL TENSION BETWEEN ANY KIND OF BRANCHING AND THE PRACTICE AND THEORY OF CONTINUOUS INTEGRATION IN GENERAL AND CONTINUOUS DELIVERY SPECIFICALLY

Obviously, there are some things that many developers would be familiar with in moving down that road, the basic things like unit-test coverage and perhaps acceptance-test coverage. How do you go further? What are some of the ways that you can verify more than you get with these unit-acceptance tests?

I think a lot of it is just not trusting one's own judgment and retaining that skeptical mind, seeing if there's something that can go wrong. My recent background has been in the area of high-performance finance where we have been writing very low-latency systems and performance testing is part of the pipeline, too. Every commit changes what will go in and will be validated against a series of unit tests to show that the code is doing what the developers think it should be doing. They'll be validated against a series of acceptance tests that show that the code does what the users would like it to do and they'll be validated against a series of performance tests that show that the performance characteristics of the code are good.

At LMAX, one of the companies where I was working during the course of writing the Continuous Delivery book, we validated what would classically be thought of as non-functional requirements. We would selectively kill bits of the application while it was running. All of this was under the control of an automated test and validate, so that when those components were restarted, the state of the system was coherent and consistent. I think that anything that can go wrong – the deployments of the software into production, rehearsing that, asserting that it works before you push the button to release it – all of those things need to be verifiable. The configuration of the system – that's one of the things that is commonly missed and treated as a separate second-class citizen to the algorithms that we write. But I can break your software just as quickly, maybe more quickly, by making the configuration invalid rather than changing the code.

So it's just taking a more holistic view, asking, "What are all the ways this could break?" rather than just considering the simplistic ways in which it might break on a developer's laptop.

Precisely.

Many of the poster children for continuous delivery are also doing continuous deployment. What would you say to people who don't want to push to production 30 times a week because it's not consistent for them. Are there still benefits from continuous delivery?

Yes, I think there are extensive benefits. The distinction I would make between continuous delivery and continuous deployment is that continuous delivery is developing code in a way by which your code is always fit for purpose and ready for release. That doesn't necessarily mean that you have to make the decision to release. That should be, can be, and should be a separate business decision: whether you want to release this now or later. But at any given point, you're in the position to be able to release it.

If you've made a release and you find a business-critical bug in your software, how long is it going to take you to safely, professionally go through all of the evaluations in order to deploy the corrections of that bug without cutting any corners? That's an important and valuable aspect of this. Continuous deployment, the process in which every commit on an automated basis will make it through into production is probably at the bleeding edge of this stuff right now. And many big organizations are doing this. This is how Amazon, Facebook, and Etsy and people like that are working. And it works really well for them. But it's not for everybody.

I've just mentioned companies that happen to be Web-centered. We were building a financial exchange. We couldn't do continuous deployment because it would affect our latency; it would slow down the rate at which trades were processed during the course of switching over. We were agonizing, wondering whether we could do it, and trying to think of ways of doing it but it didn't make sense for us. Maybe we could have solved that technical problem but it wasn't of value to the business at the time. And so I think that's the distinction. It's a matter of figuring out what suits the business, but working in a way that keeps the software deployable is a very, very healthy practice.

What are some of the preconditions that indicate you are mature enough or ready to start doing continuous delivery? Sometimes I'm talking with a developer who says, "Yes, we would love to continuous delivery but we don't actually write tests."

Well I think that's a precondition. If you're trying to make sure that your software is permanently in a position to be releasable into production then you have to evaluate that assertion some way. And the way in which we do that is by writing tests: automated tests. You can't afford to be doing [these tests manually]. This doesn't eliminate the possibility or even perhaps the need to do manual testing. But we don't want to use human beings for dumb, manual testing.

Human beings doing regression testing to my mind is an anti-pattern. Human beings are not very good at it, they're not effective. It's expensive, slow, and it doesn't catch as many bugs as automatic tests do. So use the automation to supplement those things. I think testing is certainly a cornerstone; automated testing is certainly a cornerstone of this practice.

I think, too, if you want to really get to the point of a viable software in the way of which I was talking about before, you need to be thinking about automating the deployment of the software because you need to be rehearsing that. And that, too, can't be a manually intensive process.

But it's kind of a journey. One of the things that I say in my presentations on this topic is that to achieve this, you need to start working in a learning environment. It's not a destination in its own right; it's a journey. You're continually improving on the process.

How do you figure out the next thing you should do in any given company? If I want to improve, how do I figure out what to work on next?

I'm not one of them, but there are several people that do some interesting work in this area. Eric Minick from Urbancode has a maturity model that describes the beginning state you know, what's kind of pathological, what's the extreme target, and the steps in between. And this is for [a grid with] a series of different practices that you would use to categorize continuous delivery and a series of maturity levels that you would [pass through], and within each cell in that grid there are practices that you can apply. Those sorts of things are very useful to all, for looking at where are we now, where would we like to be, and what are the next steps to achieve that. ▶

You mentioned an anti-pattern earlier. Are there other common anti-patterns that you see getting in the way of continuous delivery?

Yes. Continuous delivery is the kind of practice that leaks out. It's not something that only works within a development team. It changes the relationships between the development-team members. It's holistic, as I think you said earlier on. It changes the relationship between the development team and their user base and the business. And certainly in the organizations that I've worked for, it changes those relationships for the better.

I think one of the huge anti-patterns is silo-ing in organizations. In my experience, you can't achieve really high levels of quality and verifiability by throwing things over the wall. The teams need to be working very, very closely together, communicating on a daily basis, interacting on tasks. So cross-functional teams all focused on delivering high-quality software is the way to go.

Since Jez and I wrote the book about continuous delivery, it has become part of the works in the DevOps movement. And I think that's one of those silos that probably needs to have the barriers broken down a little bit. The relationship with the business, the relationship between developers and testers is another. And I think that's the biggest anti-pattern, that's the toughest object, the toughest barrier to these kinds of practices in most organizations.

One of the things a lot of companies in this space are playing with is the idea of feature toggles or flags. What are the main benefits are of that in a continuous-delivery environment?

Yes, that's an interesting question. There's a fundamental tension between any kind of branching and the practice and theory of continuous integration in general and continuous delivery specifically. Within continuous integration, what we're trying to do is that we're trying to get the earliest feedback that we can, that my changes work in the context of everybody else's changes. Whether it's a feature toggle, a feature branch within a repository like Git, or a separate branch in your subversion repository, any branch by definition is designed to isolate change. And so there's a fundamental conflict between the idea of continuous integration and branching of any kind and that's a tough problem to get around. My own view is that you can't count yourself as doing continuous integration if you're not submitting your change to the main line, trunk head, whatever you call it, at least once a day.

I do training with GitHub and one of the things we talk about is feature or topic branches and I think in many ways for companies that haven't started to use them, they're good step forward. The challenge is that once a company does start to use them, they have 16-month-long feature branches and wonder why the last person to integrate at the end of the month has problems.

Yes, precisely.

I believe Michael Fowler made one distinction, which is that having one long-running feature branch isn't too bad. It's when you have two of them that you generally run into integration as long as you're rebasing.

Yes. If you keep touching head, that's better than not, but still you're fundamentally separate. And going back to your question about feature toggles, they're doing the same thing in a sense. But for me, it's kind of a slightly healthier way forward because you can choose your evaluation.

I know different companies take different strategies – Etsy, for example. When they're running their tests I believe that they run them against the production version of the feature toggle, which means that those features are not being evaluated in those chains so there's a danger then that they might get a big shock later on when they turn these features on and they don't integrate with the rest of the features.

The other option that I've seen is that people will flip the feature toggles the other way so that in tests, they'll be running with the as-yet-unreleased features turned on so that they can assert that the features still work with the rest of the code – but in production those features will be disabled. And again, then, your testing is slightly faking the environment for your production.

I think it comes down to that there isn't a perfect answer to this because of this fundamental tension. And so it's a matter of judgment and it's a matter of project priorities to determine which mechanism works best.

And it seems like each solution becomes the new problem.

Yes.

Feature branches are great because it means you can release at any time from master but now you've got un-integrated work. Feature toggles mean that at least you're testing that your code doesn't have meaningful semantic conflicts and that it compiles with everyone but now you get this combinatorial explosion. I want to run my acceptance tests against every possible combination of feature flags.

Yes.

And that becomes its own separate issue. On the other hand, there was somebody from Etsy speaking yesterday who said, "Yes, we have hundreds of feature flags and so far it's not been a problem."

Yes. Clearly it works for them, so they made the right choice. I mean, they're not having problems with that. I think that either of these can work. What is always going to be painful are long-running branches that are staying away from head so you need to keep merging, whatever your branching strategy is. But certainly one of the keys of that is to branch as late as possible and the branches should live for as short a time as possible.

One of Etsy's engineer managers gave a presentation called "Screwing Up For Less". One thing he talked about was that testing is important but that other side of the coin is measurement. At some point, your tests aren't going to catch everything so how do you reduce the costs of failure?

This is very close to my heart in terms of the appeal of the scientific method for me. I want to [push this] and companies like Etsy are showing the way forward. They're experimenting in production and evaluating at the level of business change. Is this idea better than this other idea? Capturing data and looking at those things, I think that's the way forward. As software developers, what we're looking for is to have an idea, get it into production, and put it into the hands of users. Evaluating the value of that is surely where all we want to be and that's a healthy place for businesses.

It feels like this supports the lean-startup methodology: the short build-measure-learn cycle, the idea of validating with cohort analytics, and things like that. Do you think that continuous delivery and lean startup are sympathetic or supporting approaches?

Yes, I do. I mean I think that they come from similar roots. My analysis is that this is the application of the scientific method to software development. Lean manufacturing, lean processes – they came from the same root, a conscious lift from "form a hypothesis, design an experiment, carry out the experiment, iterate". They're all the same thing and the scientific method is the most important invention in human history. It's the best way of solving problems and it's the most effective way of solving problems so it's kind of inevitable that there will be parallel evolutions of these ideas. That's the best way forward. So, I see continuous delivery and lean thinking as very well aligned in terms of approaches.

In a lot of organizations, you're breaking things, you're making mistakes. You may break production; this is not going to be a surprise. Are there any things you've noticed that have helped to change cultures where the default environment is "Wait, you took production down, you're fired"?

I'm not quite sure how to answer that question. I can give you a fun war story from one of the companies at which I've worked, where these ideas have kind of become prevalent across the company. We had a new starter, in the ops team, who went into the server room and unplugged one of the servers and caused us some problems. He was scared to death that he was going to get fired. I thought the response was perfect: "We're not going to fire you. You've just learned this most valuable lesson. You're never going to do that again, are you?"

I think experimentation is fundamental and part of experimentation is that sometimes things are going to go wrong. The trick is to try and limit the impact of the experimentation and to do the serious, dangerous experiments in safe places. So think about maybe staged rollouts: you roll at a small proportion of your user base. That's one technique. Or think about the sorts of high levels of testing that I've been talking about: carry out your experiments in a live-like but not a live environment. It's those sorts of things. Crossing your fingers and hoping is not a good way forward. ■

Delivering Continuous Delivery Continuously

Simon Hildrew started out as a Java developer but was sucked into operations at a startup. Later on, after heading up the operations team at the Guardian for a couple of years, he set about solving some of the challenges of an organization with an ever increasing desire to adopt a DevOps culture.

Simon Hildrew, infrastructure developer at the Guardian, spoke at CRAFT 2014 about the media site's continuous-delivery initiative: how it started, the approach followed, the cultural challenges, and the technology needed to make it work for them. This is a summarized transcript of the presentation.

The Guardian's monthly unique users grew dramatically from 1 million in 1999 to 100 million in 2014. We have about 15 software products that we deploy on a regular basis. Those are made up of around 100 deployable components.

Deployment history

First, a bit of history on how we used to deploy. In 1996, when we first put Web servers on the Internet, it was a manual process.

By 2002, we had graduated to copy-and-paste from staging to production. By 2004, we used Bash scripts, which got more complicated by 2006. In 2007, our deployments were infrequent enough that each one of them was celebrated with a cake, about once every two or three weeks.

In 2010, we decided to try and rewrite Bash scripts in Python. We also did a lot more automated health checks and there were far fewer manual processes involved.

Around this time, we were doing about 100 production deployments a year, and that growth was largely due to the fact that we had more products rather than deploying more frequently.

In 2012, the number of deployments had grown to about 300. This was the point at which it was getting quite frustrating and difficult because ▶

it cost Operations a lot more time. We had to hire a full-time release manager to coordinate all of it.

So we rewrote our scripts again, and this time we did it in Scala because it was very popular at the time; it was the shiny, new thing. A couple of developers did it, and they had a rule from the get-go, that there would be no manual intervention on the deploy process from beginning to end.

It was a command-line tool at that time. We then wrote a Web application that sat on top of it to kind of drive it all, and added auditing and other features.

After 2012, we really ramped up the number of deploys we could do. In 2013, we managed to deploy over 10,000 times into production – 40,000 times counting all our environments.

Deployment as a service

I'm going to add one more point to Martin Fowler's definition of continuous delivery, which is that anyone can integrate the deployment service into their own tooling, their own monitoring, and their own pipelines. Call it "deployment as a service", if you like.

One of the main motivations behind this approach was that the Operations team was the gatekeeper to production. Developers and QA would come over to Operations and say, "We're happy. We want to put this version into production now."

After finishing the deployment, Operations had no idea whether it worked or not. They didn't really know what product they were deploying and relied on developers to check for success or failure.

When developers can deploy for themselves, Operations doesn't need to get in the way. However, Operations needs to know about deployments in case of outages. If Operations doesn't know a deployment happened, it can take much longer to fix problems.

Another motivation factor was poor business feedback during staging. At the Guardian, QAs and devs would put something in the staging environment and ask the editorial staff, "Is this what you wanted? Does this look how you want it to look?"

The editorial guys would say yes during staging but suddenly change their minds after deployment to production. This happened when they were confronted with the real dataset in a real-life environment where they'd be using these tools in anger. So you need to deploy into production before business people can really say yes or no.

There's also a reliability argument. The more often you deploy, the smaller the things that you're pushing out. Thus you're also gaining more confidence in your deployment mechanisms.

Tooling

For reference, we use GitHub as our source repo for the most part, TeamCity as the continuous-integration environment, and Riff Raff as our in-house deployment tool.

Riff Raff is basically a Web front end for the Scala deployment scripts mentioned earlier. It's written in Scala, the Play 2 framework, and it's got a Mongo database behind it.

It's a tool that does one thing very well: push-button deployments. And it's extremely simple to use. Around that, it keeps audit history that records who pushed the button and when. It has made continuous deployment very easy.

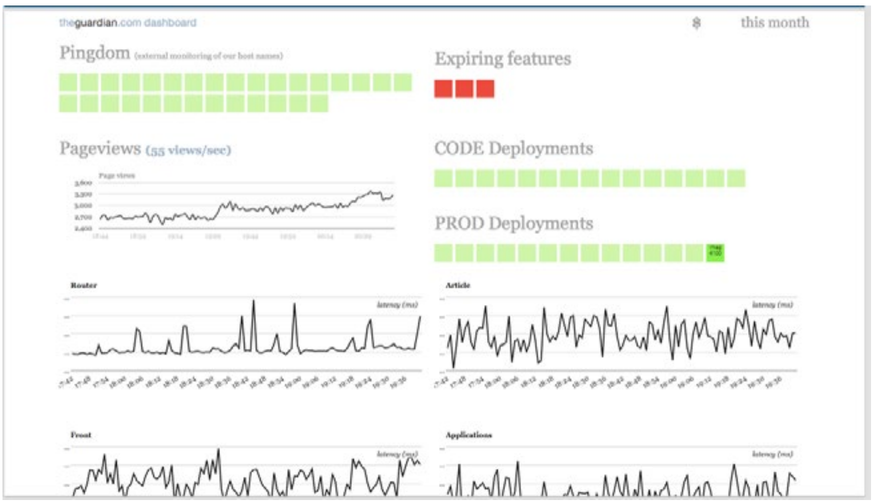
Riff Raff is able to handle HTTP callbacks to notify once a deployment is completed in a particular environment. Its API allows anyone to do pretty much anything that can be done through the interface.

It's also agnostic regarding deployment approach, which is fairly important when you have legacy environments and multiple ways of deploying code.

Deployment approach

One of our old-school deployment approaches is to copy a JAR or WAR file over to a server and restart an application. We have some Django apps as well that we deploy via copying artifacts over and restarting the Web server.

Auto-scaling deployments is becoming our de facto deployment approach these days: scale up to an auto-scaling group, let the new instances come in, download the new code, put it into service, and then just



kill the old instances. We use Fastly as our content delivery network.

Riff Raff provides an audit trail for all these approaches. More recently, we've added the ability to install RPMs and also to apply AWS CloudFormation templates.

The above is a dashboard for the main guardian.com website. This is the mobile and responsive site that's coming up. On here, you can see a lot of availability metrics and page use and all that kind of stuff.

On the right side, you can see the status of production deployments and code deployments. The boxes represent each of the individual components and the color indicates whether the last deployment for that component was successful or not. If a box is red or yellow then someone needs to look into it.

We also heavily use feature switches. One of the things we've added recently is that all the feature flags in this particular product have an expiration date. If a flag expires, you can't deploy anymore. You either have to extend it or get rid of the feature flag entirely - or get rid of the feature entirely if it's no longer being used.

Deliverator, above, is a tool for tracking what's going on in the different development environments: code, release, and production. On the right side you've got a list of the GitHub pull requests that got pushed into production on the latest deployment.

Under "Waiting in master", you can see the pull requests that have been merged into master but not yet deployed into production. "Under review" shows a couple of pull requests that are ongoing and where they've been deployed. At the push of a button, one can deploy them into any environment.

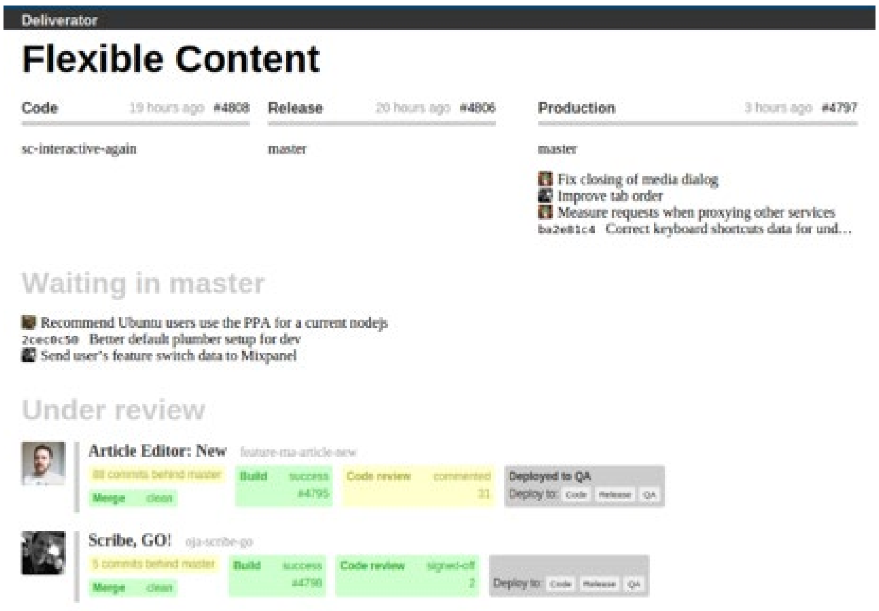
We have a mix of hardware with private (OpenStack) and public (AWS) clouds. We needed a tool that could tell us where everything is in all those environments. We built that into our Riff Raff API. Developers started writing tools that query the API to find the Web servers where their application is running so they can hop onto them.

Processes

So what are the processes that we introduced to help us along with this?

First of all, software should be deployable throughout its lifecycle. This is one of the things that we do very quickly at the beginning of any new project ►

GET THE MOST IMPORTANT PRODUCTS TO THE BUSINESS MOVED OVER BECAUSE IF THEY SEE THAT YOU CAN GET A BUG FIX OUT WITHIN AN HOUR OR EVEN HALF-AN-HOUR, THEN ACTUALLY THAT BUYS YOU A LOT OF LEVERAGE.


Deliverator

Flexible Content

Code 19 hours ago #4808 — sc-interactive-again

Release 20 hours ago #4806 — master

Production 3 hours ago #4797 — master
- Fix closing of media dialog
- Improve tab order
- Measure requests when proxying other services
- ba2e81c4 Correct keyboard shortcuts data for und…

Waiting in master
- Recommend Ubuntu users use the PPA for a current nodejs
- 2cec0c50 Better default plumber setup for dev
- Send user's feature switch data to Mixpanel

Under review

Article Editor: New feature-ma-article-new
88 commits behind master | Merge clean | Build success #4795 | Code review commented 31 | Deployed to QA, Deploy to: Code Release QA

Scribe, GO! oja-scribe-go
5 commits behind master | Merge clean | Build success #4798 | Code review signed-off 2 | Deploy to: Code Release QA

or any new component, in this order:

1. Set up a GitHub repository.
2. Set up a continuous integration build in TeamCity.
3. Make sure it's deployable. As long as your build in TeamCity is spitting out a correctly formatted artifact or zip file (containing a JSON deployment file), you can deploy it.

The JSON deployment file defines how to deploy the build, what deployment approach should be used, where to find the artifacts, etc.

Secondly, we prioritize keeping software deployable over new features. Feature branches and switches work well for this. We use switches for features that require coordination between teams so we can disconnect deployment from actually turning it on in production.

As an example, the code for last year's domain switch from guardian.co.uk to theguardian.com had been in production for months before the actual switch.

Thirdly, we have fast and automated feedback on the production readiness of any changes that we push out. There are a bunch of tests during the continuous-integration stage. We will push a new build into staging once it passes those tests. Then, there will be some smoke testing or integration testing against those staging environments.

Resulting changes

Adopting continuous delivery had a cultural knock-on impact to the way we work at the Guardian. Fundamentally, it's been a shift of roles for Dev, QA, and Ops.

From their point of view, the devs moved from being just product developers to doing a lot of operations and support. They own the production environment as well as the code.

QA people have moved from a lot of manual regression testing to focusing on test automation. The manual testing that we do now, together with the UX guys, is to provide feedback to the developers on how to improve their work.

Operations has moved from a gatekeeper role of dictating what does and doesn't go into production to becoming much more of a watchman and an overseer and sometimes an auditor, as well. They're watching what is happening in production but they're not directly involved in the day-to-day running of the products.

Operations is also used quite heavily as a consultant resource as well. So if a dev doesn't really understand what they're doing operationally, they can come and ask for support from Ops for half-a-day or a day.

How to start

We experimented a lot. Three or four years ago, we started breaking out some of the components of our massive, monolithic Web site to run those in little platform-as-a-service (PaaS) tools. So when you hit the front page of our Web site, you fetch the main site and in the background you also fetch four or five different components that are distributed in various PaaS.

Heroku, App Engine, and Elastic Beanstalk are all good PaaS. We were using Google's App Engine and the devs were looking after them. It didn't matter as much if the components fell over because it wasn't a monolithic architecture anymore. This really helped us to move towards the mindset that developers can look after products, and it's not that scary after all.

All of these platforms have simple ways of doing continuous delivery. It's there out of the box, so it's easy to integrate – especially when you're working with little applications that are easy to test. At some point, you will need to move to something bigger but I recommend that you start small.

For our new mobile Web site we applied continuous delivery from the get-go. It worked really well and the mobile team then evangelized other teams to start using the new deployment tool to be able to deploy by themselves.

We also got external people spending some time with us and saying to the QAs that these are the advantages for us, helping them to start moving to that new way of thinking.

One important thing is that you need to make any tools you put in easy to use. If you can't just hit the Web page, type in the bare minimum, and hit "go", then it's too complex.

The last thing I would recommend is to follow the business priorities. Get the most important products to the business moved over because if they see that you can get a bug fix out within an hour or even half an hour, that buys you a lot of leverage. It allows you to expend time and money in other areas and get other products onto it.

Ongoing debates

When we first started doing continuous delivery, I assumed that every product would be continuously deployed all the time.

But that is hard for a Web site or other complex products. It is much harder for a computer to spot anomalies than it is for humans who check it for 10 minutes after the deployment. Even after a deployment is completed, it may not necessarily be successful.

The other thing is that people need to own the production system. They need to actually push the button to deploy into production.

One example of this was a component in the mobile Web site that was set up to deploy continuously. It broke after three weeks. They thought they had shipped a lot of features in that time, but it took two weeks for someone to notice that the code wasn't actually making it into the production environment.

Another ongoing debate is about cost and value. This was an expensive endeavor indeed. It took about a year to write the Web app and get to a point where it was as easy to use as it is today. It took about six months in terms of dev effort to migrate all the products that we have onto it.

However, there was also a massive cost in not adopting this approach. If we hadn't gone ahead then we would still need a full-time release manager, and between a half and a full-time Operations person doing deploys all the time.

Also, we used to spend two days out of every 10 doing regression testing. That kept us stuck in a two-week release cycle. That also meant that we were thrashing between bug fixing and features. The devs had already finished all the features when they had to switch back to dealing with bugs.

The final debate is technology versus culture. I've heard a number of talks that say continuous delivery is all about changing culture, but I've come from a place where I would say that continuous delivery was all about the technology.

Fundamentally, I think it's both. If you have the technology without culture, you can still release every two weeks and not necessarily gain any of the benefits that you would have otherwise.

That applies the other way as well. If you have the culture but not the technology, then you're going to have a lot of friction between what you're trying to do and what is possible. So it needs to be a carefully choreographed dance of all of these things in order to get it working really well. ■

Rachel Laycock on Continuous Delivery

Rachel Laycock works for ThoughtWorks as a Lead Consultant with 10 years of experience in systems development. She has worked on a wide range of technologies and the integration of many disparate systems. Since working at ThoughtWorks, Rachel has coached teams on Agile and Continuous Delivery technical practices and has played the role of coach, trainer, technical lead, architect, and developer.

Interview with **Rachel Laycock** by **Graham Lee**

Graham Lee discussed continuous delivery with Rachel Laycock, lead consultant at ThoughtWorks in Johannesburg, at QCon New York 2013.

You gave a talk on polyglot architectures for rapid release. Can you tell us how that topic came about and what you were thinking when you created this talk?

Sure. ThoughtWorks has been doing continuous delivery for a few years now. I know that Dave Farley wrote the book with Jez Humble, who still works with us, and we sell that as something we can provide for our clients.

I went to one client, thinking, "This is awesome. I am finally going to do DevOps stuff. Maybe play with Puppet or Chef. I really get to understand PowerShell." When I got on to the client and we started looking at which parts of [the software] we could break off, start putting tests around, and release in smaller pieces, we realized very quickly that the architecture that they had wouldn't support that at all. They'd created this huge cyclic-dependency mess that was going to be really, really hard to pull apart.

The majority of the time that I spent on that project was just figuring out where we could create seams and really educating people in the organization about their architecture and how they can think about architecting not just for build-and-run but also for release. If you want to be able to deploy regularly, you need to break things into small enough pieces that you can deploy them regularly. You also need to have a good understanding of your dependencies and, if you have things with long release schedules, of whether you can break out that dependency or create your own component that does something similar so that you're not tied to the release schedule because, essentially, you are going to be tied to the longest release schedule of any dependency that you have.

So the "big ball of mud" architecture was what was stopping them from being able to adopt this? Or were there other factors, too?

There were other factors. One of the other factors was that one of the reasons that they'd created this architecture in this way was that they treated their developers very much as fungible resources. They were just developers that were put on any project at any given time, so they created their architecture in so that it was all written in exactly the same way – which means that they were just creating tons and tons of dependencies across the projects. A developer would be on a project and would write something over here and then be on another project. Since it was very much about building as quickly as possible, the developer would create some dependency to something that was completely unrelated and this would continue as developers got moved around.

It was very much the decisions of the architects. The developers just sat there and hooked things together and wrote the code. And a lot of it was generated code. So the architects have created this generated architecture and the developers had to just hook things together.

Because of that context and the environment that they were working in, the developers weren't really thinking about things in terms of "How is this all fitting together?" or "When I use this dependency, what does it mean?" They were just thinking and were very much pushed in the direction of "Just build something. Just build it."

It sounds like changing the mindset of the developers and the architects was going to be key in getting any kind of technological change out of this system.

It was – changing the mindset of the developers, the testers, the architects, and pretty much every manager that was involved in some way and changing the way they thought about it. They often used to ask me, "Can we do continuous delivery without agile?"

I said, "Yes, sure, but you don't have to do a small analysis just in time. What you do need to do is to have your developers and testers working together because in order to have continuous delivery, you need to have a CI, you need to have a build-and-release pipeline, and in order to trust that, you have to have tests in it." You have to move away from the mindset of "Developers write some code and the testers are going to check it at the end."

That's what I consider to be quality control. They tell you at the end how buggy you are or how good or how bad your code is and give it a mark out of 10, and then come back and say you need to get seven to pass. With continuous delivery, it very much relies on the quality you build into the product either through unit tests – which for the majority of time, I would say you want huge unit test coverage – but also with integration tests. Whether it's contracts between the system, contracts between your own components, functional tests, or especially in the case of unit tests, it's better to write those tests up front to use the TDD approach. When you are using the TDD approach, you are also thinking about the design and you are really building that quality in, and that's what I consider to be quality assurance.

And in order to do that, you need to have very good communication between the testers and the developers and you have to change the way the developers are writing the code. So these are fundamentally people changes and not technological changes.

The technological bit is sometimes hard but it's often a lot easier to solve, especially with the tools that we have now. We've got so many awesome CI servers. Obviously ThoughtWorks has Go, but there are plenty of others out there. You've got awesome tools for virtualization, infrastructure automation, and all that configuration stuff. And we even have the capability to test those now with tools like Puppet and Chef and PowerShell. There are things continuously happening in that space. Even in the Windows space, which is where I spend most of my time, the support for some of those tools is getting better and better all the time.

So the tooling is there, but they very much rely on that being reliable. [As the code is tested,] do I trust what's coming through? Because you can just put your entire ball of mud through the pipeline but it doesn't mean it's going to work at the end. You need to be assured of that quality all the way through and that comes all the way back to the development process, which is how your developers are writing the code and how they are communicating with the testers. So do you need to do every part of agile? Do you have to do a stand up? Do you have to do this? Probably not, but these are things that really help you. For me, fundamentally, it's that communication between the developers and the testers. And then if you really want to do continuous delivery, you involve your ops guys as well.

If you are going to move towards TDD then you are perhaps changing the power balance of the system. You now have the developers writing the tests. Does this make the testers or even the developers uncomfortable?

I wouldn't necessarily say that it's changing their power dynamic. I think that it really depends on the organization and where the power existed in the first place. We like to forget that organizations are politically driven and they are all about people, and we as developers often think that we want to focus on the technical solutions. Of course, we should write tests. We are building quality in the code but the power dynamic is really dependent on the organization.

What I have noticed is that the change is really in the resistance to change. The developers are used to just writing the code and being done with it, and the testers are going to find their bugs, and they can just move to the next story. But in the TDD approach, they have to think about how they are going to build it and about what tests they can write, and use that to drive the design.

So they do have to do more work and especially initially, when they are learning, it does take longer, and they are resistant to that. I know that at QCon and ThoughtWorks, a lot of developers love learning new languages and new tools and we get excited by that, but a lot of clients and a lot of developers like to be masters of what they know. When you introduce something that they don't know, they get very uncomfortable because suddenly they are not the masters anymore.

Some people are quite adaptable to that but others can be very resistant, and they can resist in a lot of different ways. Depending on where they are politically in the organization, they can pay lip service, so if this is coming from a higher order like "We must do agile transformation. We must do this project like this," then they can be like "Yeah, yeah, I'll do it, I'll do it" but they don't – they just try to get you out the door. Others might be nine-to-fivers. They just want to get in there, do their job, and want to go home. But often when they start to understand it or they start to enjoy doing it, they can totally change.

In terms of the testers, what I generally found is that the testers usually get the worse end of the stick in the development life cycle anyway, because they are right at the end and the testing always gets pushed. Especially if you are using a waterfall methodology, they are supposed to have two months of testing and now they've got only two weeks to test as much as they can. So they are frantically trying to test stuff and they come back with bugs to the developers, and the developers are obviously not happy about that. So they are getting pushed from both ends. [With TDD,] their lives improve because suddenly someone is talking to them about writing tests. Usually, we start by getting the testers to do acceptance testing and start automating some of their tests, and then we try to get the developers and testers to work together. The purpose of that is to improve those tests because those tests are usually long-running and you want them to be more performant and, guess what, developers are usually better at that.

But the second thing that I usually get to as teams mature is to get the developers to see what the testers are covering in their tests, and to communicate what we are covering in our tests, so that we are not overlapping. We usually are writing unit tests because they are much faster, and fewer of those end-to-end tests. So I have not seen the testers be resistant. I have seen developers be resistant for lots of different reasons ¬– as I said, it depends on the organization.

In another example, the client was trying to get some software out of the door for such a long time and the developers were just fixing bugs all the time. The client asked, "Can you guys come in and fix the developers?" Their developers were open to learning TDD or any of these things but said, "What's the point? Because we are going to get pushed again and told, 'We just need to get stuff done.'"

[We said,] "Oh, that's an easy question to answer. Your managers brought us in so you might as well use us while we are here."

When the product owner, the manager, and the dev lead saw that the developers improving so much and the number of bugs go down – as opposed to fixing and creating more bugs – they started to ask "What can we do?"

We said, "Well, you know when you keep disrupting things and making changes in the middle of the iteration? That's kind of annoying. It doesn't actually help and it puts a lot of pressure on people so people don't do the best job."

What I have quickly discovered through consulting is that it's never the technical problem. It's never that we don't have the right tools. In my talk, I said that if you are going to choose a tool, if you are going to choose a new language or you want to go polyglot for whatever reason, I am okay with it as long as you can test it, and the tools are mature enough to allow you to test it, put it into the build, and deploy pipeline. Somebody writes tests for pretty much all languages as soon as they are written. So the tools are there. All the tools for CI are there. There are so many. ThoughtWorks has one – Go – which is awesome. There are the tools for the build configuration and for the virtualization. Those problems have been solved.

The problem that you have to solve is the context of the environment that you are in and that's the people: what are their goals, what are they trying to get out of it, how are you going to handle their resistance? Depending on that, they can either transform and learn some of these techniques or they can't, because if people want to, they will learn it. If they keep resisting, they won't learn it because they have already decided "I am not going to learn it" or "I can't learn this" or whatever it is.

Are there patterns of developers' resistance? Are there particular categories? How can we identify people who fit these patterns and motivate them to adopt these changes?

That is a really interesting question because I have never thought about it and the minute you said that, I thought "I could write a book about this." But I do not think it's a pattern of developers' resistance. I am sure there are a lot of psychology/people-related books out there that we can read and learn from in order to figure how we can help people change, how we can figure out what their resistance is, and how to, maybe not counteract that, but how we can say "Hey, you are resisting because of this, but this is the benefit."

I am sure there are patterns but for me, it's more instinctive, it's more about people. I just look at them as people and think about why they are resisting. Is it because they are uncomfortable and they don't feel like masters? Then I say, "Hey, your secret is safe with me. I am not going to tell anyone that you didn't know what you were doing for six months. That's why I am here. I am going to shield you from all that and if the tests are rubbish then that's my fault because I am the one that is teaching you."

How you can handle it really depends on what they are resisting. I am sure there are patterns, but am I going with what this person wants? And that assumption about people being nine-to-fivers? I've made that assumption in the past – that guy is a nine-to-fiver so he is not going to want to learn – and then dug in and got a little history. And it turns out that for the project that he was on last year, everyone was working so hard that they were sleeping on their desks, so, yes, now he is on a nice, easy project and he wants to work nine to five. Fair enough. That doesn't mean he doesn't enjoy what he is doing and not want to do it. So for me, it's always a question of finding out what motivates that person or why they are resistant because there is usually a counter-argument or a way of convincing them to do the right thing. And telling most people, especially if they want to be masters, that if you start writing tests in your code and you are creating quality in your code then you will become even more of a master can sometimes win them over as well.

You talked about how perhaps developers have to change. There is a famous trope in software development that your architecture will eventually reflect your org chart, so there is presumably organizational change too, and does that lead to manager resistance?

Very, very, much so. Changing the developers and the testers is easier than changing the manager's mindset, but again that can be the context of the environment. In a lot of the environments that we go into, there is a key executive who looks after hundreds of testers and another one who looks after a hundred of developers, and never the two shall meet. [We want] these teams to start working together as teams. We call them feature teams because they are basically a combination of the developers, the testers, the analyst, the PM, the user-experience person, and anybody that you need to build that ►

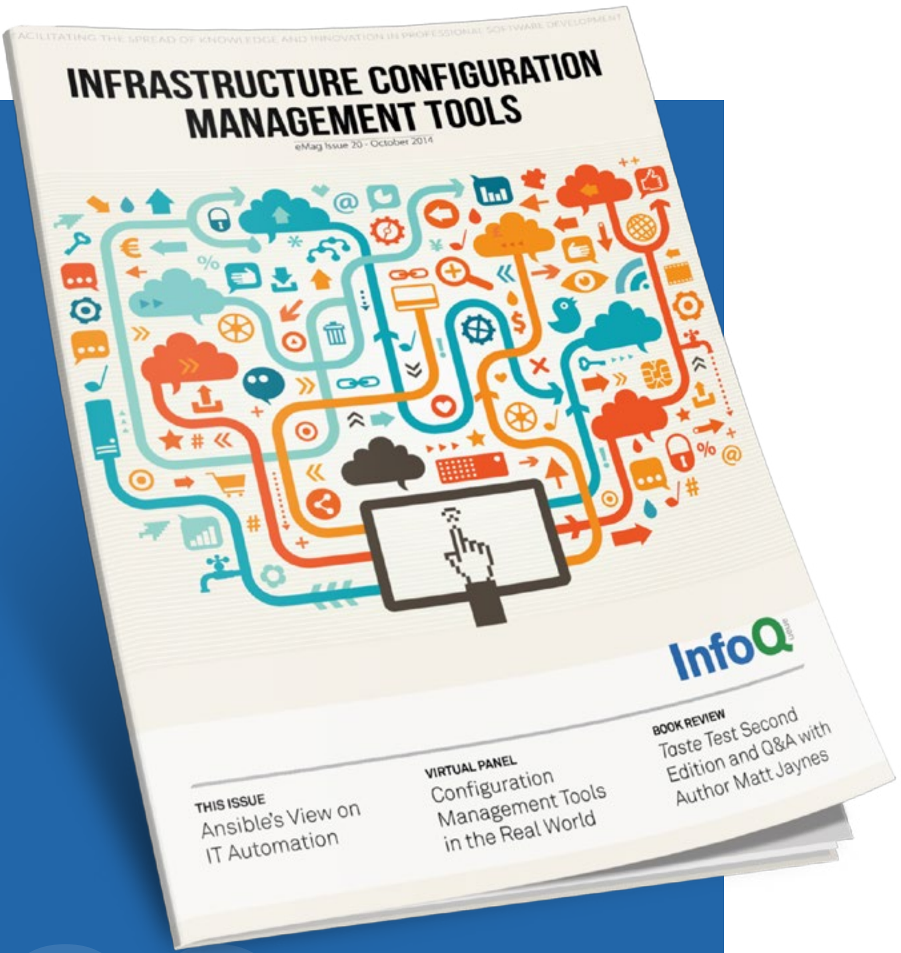
team. They should be together if possible and working together as a team, not working in silos. And that can sometimes be a really hard sell because you are asking these people to change the structure of their organization. The only way to really sell it is to prove it, so we always go with just a pilot: "Let's take a couple of these testers and some of these developers from these different environments and get them to work together and prove that this will help us release quality software regularly."

And usually when we prove that, that gets them into the mindset of "I can see this works," but that kind of change, changing the organizational structure, takes a long time. and Even to get that team together, you've got to go all the way up to the executives in each different group to convince them that this is the right idea. So there is a lot of explaining the benefits and trying to convince or influence people to try to make some of these small changes just to prove that it works.

Otherwise, and this has been proven, any point at which you have communication breakdown between teams is where you will find the complexity in the system. I have worked on projects where it's been awesome: agile and lots of TDD and technical practices. And it looks like the code is awesome. But because the communication between ops and the development teams has broken down, to release it suddenly takes forever and ever and ever. That's complexity that can only be handled by people communicating with each other, and in order to get that team together, you've got to align all different directions in an organization to convince them to let you do that. That's even harder. This stuff is hard. ■

WHAT I HAVE QUICKLY DISCOVERED THROUGH CONSULTING IS THAT IT'S NEVER THE TECHNICAL PROBLEM. IT'S NEVER THAT WE DON'T HAVE THE RIGHT TOOLS.

PREVIOUS ISSUES


FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

INFRASTRUCTURE CONFIGURATION MANAGEMENT TOOLS

eMag Issue 20 - October 2014

InfoQ

THIS ISSUE
Ansible's View on IT Automation

VIRTUAL PANEL
Configuration Management Tools in the Real World

BOOK REVIEW
Taste Test Second Edition and Q&A with Author Matt Jaynes

Automation in the Cloud and Management at Scale

InfoQ

Lessons on Cloud Management at Scale: An Interview With Randy Skopecek

Agile Project Management

InfoQ

Why the Agile Project Manager is the Secret Sauce for Development Projects

Agile Project Estimation and Planning

InfoQ

Interview with Eduardo Miranda about Estimating and Planning Agile Projects