

FASTER, SMARTER DEVOPS

eMag Issue 56 - Dec 2017



InfoQ
neue

ARTICLE

How to Deal with
COTS Products in
a DevOps World

Q&A

Merging
Agile and
DevOps

BOOK REVIEW & INTERVIEW

Designing
Delivery by
Jeff Sussna

IN THIS ISSUE

6

Faster, Smarter DevOps

Moving your release cadence from months to weeks is not just about learning Agile practices and getting some automation tools. It involves people, tooling and a transition plan.

12

The Things I Learnt About DevOps When My Car Was Engulfed by Flames

Framed in the story of the author's car catching fire, this article describes five ways of thinking to help understand DevOps culture, and behaviours necessary to create an effective DevOps team.

16

How to Deal with COTS Products in a DevOps World

The most popular agile framework, Scrum, predates the growth of DevOps, and some re-thinking is required to make the system work in a DevOps environment.

24

Merging Agile and DevOps

Mirco Hering explains why we shouldn't leave COTS products (and the people working on them) left behind in a DevOps world. With creative solutions we can apply good practices from custom software.

32

Designing Delivery Book Review and Interview

Book review and interview with Jeff Sussna, author of "Designing Delivery", on cybernetics, service exchange, customer-centric brands and a new definition of quality in a service-oriented world.

FOLLOW US



facebook.com
/InfoQ



@InfoQ



google.com
/+InfoQ



linkedin.com
company/infoq

CONTACT US

GENERAL FEEDBACK feedback@infoq.com

ADVERTISING sales@infoq.com

EDITORIAL editors@infoq.com

A LETTER FROM THE EDITOR



Manuel Pais

This DevOps eMag has a broader setting than previous editions. You might rightfully ask, “What does faster, smarter DevOps mean?” Put simply, it means any and all approaches to DevOps adoption that uncover important mechanisms or thought processes that might otherwise get submerged by the more straightforward (but equally important) automation and tooling aspects. From blending agile and DevOps to escaping cognitive biases, planning for transition, and bringing COTS into DevOps, this eMag is a potpourri of insightful practitioner advice.

Derek Weeks, vice president at Sonatype, wrote the article that inspired this eMag, in which he talks about the dangers of focusing only on improving speed of delivery by means of new tooling. Weeks highlights the importance of also defining and evolving a transition plan for an effective DevOps strategy that brings everyone on board (bottom up and top down). Shared goals and terminology across teams, adequate pipeline metrics, and quality strategy are some of the key aspects to plan for, while being explicit in our automation purposes so we can better sustain and evolve our tool chain.

John Clapham, independent coach, trainer, and consultant, shares his lessons learned from having his car engulfed in flames (!) and how that relates to DevOps. Our working environments are full to the brim with cognitive biases and knee-jerk reactions — just like Clapham was, as he was positive the smoke couldn’t possibly be coming out of his recently serviced car. In many ways, DevOps requires us to identify and explicitly act upon our instinctive “business as usual” behaviors in order to create a high-performing, blameless culture.

James Betteley and Matthew Skelton, from Skelton Thatcher Consulting, remind us that Scrum helped accelerate delivery and, in fact, contributed to the rise of the DevOps movement, as traditional operations teams became overloaded. Yet, few organizations revisited Scrum in light of this new normal, where development teams take on responsibility for running and monitoring their applications in production. They explore in their article moving from product to service backlogs, operability stories, planning for unplanned work during the sprint, and more techniques for blending agile and DevOps.

We shouldn’t exempt business-critical COTS platforms from DevOps adoption, recommends Mirco Hering. His article is chock-full of practical advice on applying version control, configuration management, and automated build/deployment even when lacking direct control over the COTS platform. Hering illustrates those practices with a real-world example of a Siebel CRM system he worked on.

Finally, Jeff Sussna’s book, *Designing Delivery*, looks at the requirements for successful, holistic service-delivery organizations. Aligning agile, DevOps, design thinking, brand engagement, and a customer-centric focus over multi-channel, 24/7 availability systems will become mandatory to survive. Read the book review and ensuing Q&A to grasp Sussna’s new way of thinking about delivery in the service economy.

These five articles provide plenty of food for thought and should trigger valuable discussions in your organization on the effectiveness of your DevOps adoption strategy!

CONTRIBUTORS



Manuel Pais

is a DevOps and Delivery Consultant, focused on teams and flow. Manuel helps organizations adopt test automation and continuous delivery, as well as understand DevOps from both technical and human perspectives. Co-curator of DevOpsTopologies.com. DevOps lead editor for InfoQ. Co-founder of DevOps Lisbon meetup. Co-author of the upcoming book "Team Guide to Software Releasability". Tweets @manupaisable



Derek Weeks

currently serves as vice president and DevOps advocate at Sonatype. In 2015, he led the largest and most comprehensive analysis of software supply-chain practices to date across 106,000 development organizations. As a 20+-year veteran of the software industry, he has advised many leading businesses on IT performance improvement practices. Weeks shares insights regularly across the social sphere at @weekstweets, LinkedIn, and online communities.



John Clapham

is an independent coach, trainer, and consultant. Offering considerable experience in continuous delivery, DevOps, and agile, he helps teams to build great products, creating an environment that is effective, productive, and enjoyable to work in. His broad experience in software development ranges from start-up to enterprise scale, formed in the publishing, telecommunications, commerce, defence, and public sector arenas.



Mirco Hering

leads Accenture's DevOps and agile practice in Asia-Pacific, with focus on agile, DevOps and continuous delivery to establish lean IT organisations. He has over 10 years experience in accelerating software delivery through innovative approaches. In the last few years, Hering has focused on scaling these approaches to large, complex environments. He is a regular speaker at conferences and shares his insights on his blog.



James Betteley

comes from a development and operations background, which is pretty handy for someone who now works in the DevOps domain! He's spent the last few years neck-deep in the world of DevOps transformation, helping a wide range of enterprise organizations use agile and DevOps principles to deliver better software faster.



Matthew Skelton

has been building, deploying, and operating commercial software systems since 1998. Co-founder of and principal consultant at Skelton Thatcher Consulting, he specialises in helping organisations to adopt and sustain good practices for building and operating software systems: continuous delivery, DevOps, aspects of ITIL, and software operability. Skelton curates the well-known DevOps team topologies pattern.



Jeff Sussna

is an internationally recognized IT consultant and design-thinking practitioner. He is known throughout the DevOps community for introducing DevOps to the importance of empathy. Jeff has nearly 30 years of experience in software development, QA, and operations, and has led projects for Fortune 500 enterprises, major technology companies, software-service startups, and media conglomerates.



Read online on InfoQ

FASTER, SMARTER DEVOPS

by **Derek Weeks**

Call it DevOps or not, if you are concerned about releasing more code faster and with a higher quality, the resulting software delivery chain and process will look and smell like DevOps. But for existing development teams, no matter what the velocity objective is, getting from here to there is not something that can be done without a plan.

Moving your release cadence from months to weeks is not just about learning agile practices and getting some automation tools. It involves people, tooling, and a transition plan. I will discuss some of the benefits and approaches to getting there.

Waterfall to agile, **agile** to continuous integration, **continuous integration** to **continuous deployment** — whatever your processes are, the theme is the same: find a way to get code to users faster without sacrificing quality. But speed and quality are sometimes in opposition. Going faster means things can break faster, and when we only think about DevOps as releases, it's easy to fall into this trap.



Established development shops cannot just jump from one flow to another. Unless you start out net new, the goal is to introduce new processes without delaying releases for three months or more and transition in lump. This is often done with a pincer approach that addresses bottom-up tactics and top-down oversight and culture at the same time.

However, because adopting DevOps tools is so easy, the trend is to focus on tactics only and adopt from the bottom up without consideration of the entire pipeline. This leads to release automation tools dictating your delivery chain for you, and not the other way around. Here are the key categories that get neglected when teams hit the accelerator without a plan in place.

Structured automation

DevOps requires automation. But what is often not considered is automation that sustains and fits into the entire delivery chain. You need to consider factors such as governance, artifacts organization and inventory, metrics, and security. If an organization establishes a vetting process for all new automation and how it fits into the pipeline's orchestration, then new automation will support what exists today and what will exist in the future.

For example, many organizations driving down the DevOps path have encountered challenges when trying to incorporate practices from security or governance teams. Historically, these teams have resided outside of the development and operations echo chambers and their processes were asynchronously aligned to the work being done. The challenge for many organizations is to determine the best ways to bring the people, processes, and technology supporting these initiatives into the fold without slowing things down. The best organizations are finding new ways to automate policies from security and governance teams by shifting away from artisanal, asynchronous approaches to synchronous processes earlier in the lifecycle.

Let's look at an example of application security. A number of technology vendors in the application-security arena are touting automation as key

value point for their solutions in order to better fit them into a DevOps tool chain. In some instances, automation means that machines are now fully responsible for monitoring, analyzing, and fixing security vulnerabilities for those applications at wire speed. In other instances, automation eases human workflows that might represent hours or days of asynchronous analysis not fit for continuous operations. In both cases, the technologies may accomplish similar ends, but their approaches could be dramatically different.

Also, one solution might be built to support asynchronous investigations by a security professional, while the other might provide synchronous support to a developer at the design and build stages of the systems development lifecycle (SDLC). Establishing a vetting process can help determine if the automation levels required by a team or process can truly be delivered before making investments. It is also worth noting that layers of obscurity frequently exist within words like “automation”, “integration”, “configuration”, and “continuous”.



Common language

Part of the reason you have so many meetings is you are not all speaking the same language. Even if all understand the other aspects of the software delivery chain, it does not mean that teams (QA, development, IT ops) speak a unified language. And the time wasted to reconcile the differences is just vapor. But the solution is easy. Be deliberate. Have a guide and agree on terminology in the tools you use up front for new aspects of the pipeline and application.

One approach I have used to establish a common language is to share a common story. In a recent meeting with a CIO, he told me his aim was to transform a diverse group of 50 people to operate under DevOps principles but that they lacked common understanding for starting the conversations. I recommended that he purchase a copy of *The Phoenix Project* for each person in the group and ask them to read it. The novel describes the efforts, challenges, setbacks, and accomplishments of a diverse group of people as they transform themselves into a DevOps practice.

For example, organizations could map their journey into DevOps similar to the way the main characters in the book do. They could discuss how their own organization might take on mastering the “Three Ways”:

- **The First Way** — understanding the flow of work from left to right as it moves from development to IT operations to the customer.
- **The Second Way** — ensuring a constant flow of feedback from right to left at all stages of the value stream.
- **The Third Way** — creating a culture that fosters continual experimentation and learning.



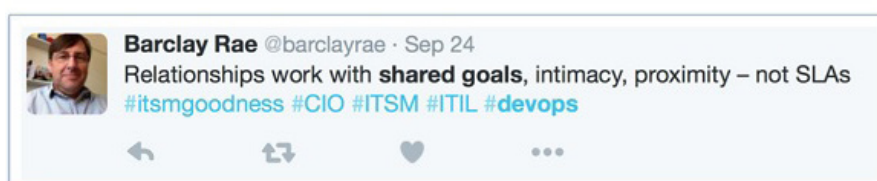
Readers can also gain a common understanding of the four types of work described in the book: business projects, internal IT projects, changes, and unplanned or recovery work.

Using such stories can provide a foundation for conversations and common vocabulary that fosters improved understanding, collaboration, and planning for the journey ahead.

Shared goals

If the entire team — all the players that are a part of software releases — does not share an objective, then the competing goals will lead to competing strategies and delayed releases. This results in a Frankenstein pipeline, for example:

- If development is not accountable for bugs in QA, they will commit features faster, but releases are slowed because of new issues. Conversely, if there is a dedicated QA team that is rewarded for the quantity of issues found or closed, unnecessary work may be unduly entering the system.
- If IT operations is not motivated by release frequency, they won't consider things like full-stack deployments due to perceived risk.
- If security teams rely on automated testing that takes four hours to complete yet the development teams are pushing out new releases every hour, some application security checks may never happen.



Pipeline and business metrics

If you do not measure the release process (builds, speed, deploys, bugs, etc.), there is no way to know what to change to increase the speed and quality of code. But pipeline metrics are not straightforward — and involve the entire team. The beauty of modern tooling is that it can collect the data and metrics

for you. But it is up to the team to decide what is valuable. Metrics available from tools or measured processes can provide common ground for understanding what work is being done or what results are being achieved. Share metrics that track work in progress, deployment frequency, lead time for changes, mean time to recover, and things that matter most to the customer you serve. Tracking and sharing these metrics can help organizations better understand the constraints within the systems and practices they are trying to optimize.

That said, you need to focus on the right metrics, which that drive business success. Metrics can help you see if you are progressing on the things your business and customers care about most — seeing the big picture. Deming called it “appreciation of the system”. For example, here are some pipeline and business metrics to consider:

- Are you completing activities (e.g., releasing builds, shipping new features, enforcing quality, checking security, responding to inquiries) fast enough to matter?
- What percentage of time are you spending on innovation compared to maintenance/rework?
- What percentage of time is spent on manual/asynchronous versus automated/continuous activity across your development and operations processes?
- Is your user base expanding or contracting? Are your customers investing more in your solutions over time?
- Are your customers getting what they want in the time they expect?



- Are you spending more or less to acquire new customers or support existing customers?

Quality Strategy

Quality is not an afterthought. The best development operations make quality everyone's responsibility and make QA a strategy and automation practice, not an execution one. Then quality goes far beyond regression testing and standard unit tests. It also includes:

- quality of the pipeline,
- component quality and security (e.g., binaries, images, tools),
- [test-driven development \(testing new functionality before it's implemented\)](#), and

- [behavior-driven testing and development](#)

Constantly improving the ways that you catch bugs means your test coverage is always increasing and the number of bugs is decreasing. It also means, if you have a dedicated QA team, that this team has open communication with IT and developers and that it has a seat at the table when it comes time to talk about quality.

Plans, hurdles, and wins

The best way to get from slow to fast consists of these practices:

1. **Have a plan.** Cliché, huh? I am not talking about a 10-page document that you put in a drawer. Build a lean

model that covers the problem, the path to solution, and clear goals. It should not be static, but built in an agile-type way. Think strategy as code — perhaps taking a [walking skeleton](#) approach. Until you talk/think through the entire flow, it's nearly impossible to catch all the variables. And things are always more elaborate than they seem. If you have ever built a minimum viable product (MVP), you will know what I mean.

2. **Clear the hurdles.** Don't think of governance and change control only after you set up a new process. Think about it from day one. And start your automation there. Pipelines are only as fast as the slowest gate. In many organizations, that is compliance and governance. So clear your governance hurdles by bringing on tooling that can do vulnerability checks on your components, validate licensing, and maintain an audit trail of your entire build-and-release process.
3. **Socialize quick wins.** Build enthusiasm as you start to get results. Address one aspect of your delivery chain and make it better. They should be relatively low risk and high value. Once you have done so, socialize the benefits. These quick wins get everyone excited about what is possible. And give some direction on what to do next. Continuous integration — due to the great tools out there — is a place that you can start that is significant and team-wide and has a serious impact. (But don't fool yourself by just implementing Jenkins, Bamboo, or

some other CI platform. Team members should be integrating their work frequently, and each integration needs to provide rapid feedback on the quality and stability of the build.)

Your environment is as unique as the combination of all of its individual parts. There is no “one size fits all”. But ensuring attention to the categories I mentioned above can help existing operations move from where they are to a regular cycle of sprints and, eventually, to continuous delivery or even deployment.

A final tip that I will offer is to engage with your community. There are numerous [DevOps Days conferences](#) and over [1,750 DevOps meet-ups](#) around the world. Right after you finish reading this article, find one in your area and plan on attending. The networking opportunities, open-spaces agenda times, and presentations shared there can be invaluable.

But don't just stop at attending the events; use the opportunity to connect with people in your local area that have already established DevOps practices and see if they might invite you into their organizations for a half or full day of shadowing. This ap-

proach is more common than you might think across the community and can lead to valuable insight that you just can't glean from online articles or conference presentations.

In modern development, it is easier to go fast than to make sure that you have the right agility to ensure future successes. It is important to recognize that taking the time to build and execute a plan that enables the right level of agility will be more productive than developing a plan defined only for speed.



Image source: <http://devops.meetup.com>



Read online on InfoQ

KEY TAKEAWAYS

DevOps and agile encourage ways of thinking that are sometimes unnatural to us and work against our instincts.

The same ways of working are often opposed to established organisational ways of working.

It takes a conscious effort to overcome our cognitive biases to make the best of our skills, and to relate to our colleagues.

Our ability to learn and improve as people (and teams) is often hampered by a perception that failure is entirely negative. This view may arise from us or be inherited from an organisation's culture.

There are five takeaways in this article, framed in the true story of the day the author's car caught fire....

THE THINGS I LEARNT ABOUT DEVOPS WHEN MY CAR WAS ENGULFED BY FLAMES

by John Clapham

This is a true story, based on a talk from DevOps Days London 2016.

It was a gorgeous sunny spring day. My family and I were driving through my hometown of Bristol, ready for another weekend adventure. We were cruising along when my wife said quietly, "I can smell smoke."

Now, I'm a good mechanic; I used to restore classic cars. The car we were driving was modern and recently serviced. I checked the instruments, everything normal. "It must be outside," I declared confidently.

Two minutes later tentacles of smoke were curling around my ankles and my shins were getting remarkably warm.

We can learn something relevant to DevOps, and many other disciplines, from this experience.

Don't let your expertise be a blind spot

People who are regarded as experts in a field are likely to be called upon to provide answers quickly. Humans are very good at this. Instinctive answers engage what has been called “[System 1](#)”, a rapid-fire but slightly lazy part of the brain. It tends to reach for the easiest answer to hand, useful in a fight-or-flight situation, less helpful in a meeting of minds. System 1 is designed to front our System 2, which is relatively rational but needs time to figure things out. The relationship between Systems 1 and 2 is like the relationship between a cache and a server request. Sometimes though, when our mental cache can't find an item, it offers the quickest thing it can find instead of calling the server. Many times, our quick, instinctive answer, based on years of experience, will be correct. Other times, probably when it's most embarrassing, it won't.

In problem-solving situations and thought work, we need to learn to anticipate and reflect on this initial, instinctive answer. A useful technique is to challenge yourself to look for a second answer, to try to refute your first. As we leap to the nearest conclusion, we often ignore useful input from others, particularly if we regard them as less knowledgeable than ourselves.

The [curse of knowledge](#) speaks of a similar challenge: how to relate to people new to a field. I believe this “curse of being good at stuff” is similar and more focused on the tendency to jump to conclusions when we feel pressure.

We swiftly pull the car over. We are in one of Bristol's less than salubrious areas. In fact, it has a reputation for tracksuits, hoodies, aggressive dogs, caps, and numerous police vehicles. I step out of the car and I'm immediately confronted by a gang of youths who say, “Get your &^%*&^% car out of here. You can't park there.”

I reply, “Look, my car is on fire. I want to get my family out and then we can talk.” The change in attitude was astonishing. “I'll fetch a hose,” says one. “Do you need a hand?” says another.

What can be learnt from this? As DevOps practitioners, we often behave in ways that seem unconventional, baffling, or threatening to others. From outside the car, the lads couldn't see the fire so they couldn't guess our context. What they saw was an ordinary car roar into their space and a stressed stranger jump out. I took time to explain the situation and they listened. Once they understood the situation (and my motivations), they made a judgement call to help.

People are more likely to assist when they understand your motivations

This is something I see frequently when people are promoting new techniques, be it agile, DevOps, or a fresh technical approach. The new concept makes perfect sense to the promoter — in fact, its merits are so obvious that the promoter doesn't explain context and motivation to potential adopters. Unable to understand or develop empathy, people will often resist or blithely continue with their habits.

The value of this kind of empathy cannot be understated. It applies both ways: if we as promoters of change expect to be understood

Ancillary systems,
with their
ownership sitting
in the uncertain
space between
Dev and Ops, are
instrumental to
rapid recovery
[from disaster].

We should also inspect our own reaction to failure. Do you look upon them as opportunities to learn? Or are they explained away with self-comforting logic?

and listened to, then we too must understand the feelings and challenges of others.

So the family are clear of the car, the heat has intensified, and flames are coming out of the bonnet. It's time to put the fire out.

I go to the boot where the extinguisher is stored. It doesn't open. The molten blobs of electrical insulation under the car provide a clue as to why. I take a deep breath, dive into the car, and grab the extinguisher. With the flames growing, I unwrap the extinguisher and leap into action — by reading the instructions. I fiddle with the pin and point the extinguisher, but the foam isn't going in the right direction. Just as I get it to spray in the right direction, it sputters and stops.

My fruitless attempts to use a tool with which I'm not familiar at the time it most matters tell us something about our approach to unexpected situations:

Don't just plan for disaster but expect it, practice for it

Operations groups are pretty good at planning for disaster, and disaster-recovery plans are recognised good practice. This calls to mind the classic boxing quote: "Everyone has a plan until they get punched in the face." Typically, efforts focus on production systems. These alone are often not enough. Ancillary systems, with their ownership sitting in the uncertain space between development and operations, are instrumental to rapid recovery. There are plenty of stories of systems recovering their compute, but being unable to recover further due to their configuration-management or deployment-tool servers not being available. The duration of GitHub's January 2016 outage was increased by [the loss of their](#)

[chat servers](#), something they relied on to understand system state and to collaborate in the event of an emergency.

We call the fire brigade, which thankfully arrives promptly. The team of four exchange very few words yet move swiftly with a sense of purpose, each member contributing in a different way. Hoses are selected, traffic is redirected, a water supply is found, people are directed to a safe distance, and the hoses are turned on. My car is drenched in gallons of water. It's clear that even if I had practised, all the fire extinguisher would have done is buy us time. "You need to understand the fire, see?" says one of the fire fighters. "It's deep in the bulkhead, very hard to reach."

So how are fire fighters so good at what they do? For one thing, they follow the previous advice: they practice for disaster and they are well rehearsed. Fire fighters also learn quickly. One way they learn is through investigation of failures.

Failures are rich in learning

The approach of Western cultures to failures appears to need a massive rethink. The DevOps and agile movements lead the way, encouraging recognition that triumphs and failure alike are opportunities to learn and improve. You are likely to be familiar with root-cause analysis, retrospectives, and post mortems (blameless and otherwise). These all encourage learning from recent events and commitment to making improvements.

We should also inspect our own reaction to failures. Do you look upon them as opportunities to learn? Or are they explained away with self-comforting logic? I often think of Danny MacAskill, a Scot-

tish trials bike rider. He practices the same jump over and over, falling hard on many occasions, until he gets it right. Each time, each iteration, brings him closer to his goal.

The approach organisations take to failure is so significant that [Westrum's typology of organisational culture](#) uses the treatment of messengers (those who often bring news of failure) as an indicator of the type of culture an organisation has built. Where reporters of failure are encouraged to speak up, their concerns investigated, and corrective action taken, you'll often find a high-performing organisation.

Finally, we are back at home. We try not to think too hard about what happened, or what could have happened in very slightly different circumstances. We upload a few photos. Friends sympathise and make summer BBQ jokes.

Months later, we are contacted by an owner of the same type of car. It transpires that this model has been suffering spontaneous fires all over Europe. None of the drivers know each other, but there are enough to form a community, enough to support each other, enough to get the attention of the manufacturer, enough to engage a lawyer, enough to prompt a global manufacturer to issue a recall. All because a few people shared their photos, and cared about the cause enough to connect with each other.

If it matters, share it — you never know who'll benefit... and it could be you

Sharing is another tenet of DevOps: share knowledge, code, metrics, styles, approaches, and patterns. Often though, we share what interests us, what we've been asked to, or what process demands. I encourage what I term "deliberate sharing": the act of sharing things because they might be interesting or having your default sharing setting as public rather than private. This includes the things you

tried that didn't work as well as those that did. Science has long recognised the value of this approach and many tangential findings have led to useful applications. Blogs, kanban boards, and open meetings are all examples of sound sharing practice. All these invite serendipity and encourage others to build on what you do.

So that's the story of the day my car caught fire and the things I learnt about DevOps from the experience. It may be instructive to read those lessons once more and note that they are highly applicable outside of the IT world. This is one of the elements that draws me again and again to DevOps. Through it and the willingness of the community to share, we may learn skills that serve us well in other aspects of life.



Read online on InfoQ

KEY TAKEAWAYS

Configuration management is the basis for any good DevOps adoption as it is crucial to enable speed.

COTS products will continue to be relevant in the DevOps world as they continue to support key business functions.

Version control for COTS products requires creative solutions to identify relevant code and store it in common source-control tooling.

It is possible to significantly reduce effort by treating COTS code similar to custom code.

Four steps will make your COTS solution more manageable in the DevOps world by making it easy for COTS developers to do the right thing.

HOW TO DEAL WITH COTS PRODUCTS IN A DEVOPS WORLD

by **Mirco Hering**

The primary objective of DevOps is to increase the speed of delivery with reliable quality. To achieve this, good configuration management is crucial, as the importance of the level of control grows with higher speed of delivery (while riding a bike, you might take your hands off the handle bar once in a while, but a Formula One driver is practically glued to the steering wheel).

Yet **commercial off-the-shelf** (COTS) products often don't provide any obvious ways to manage them like you manage your custom software. This is a real challenge for large organisations that deal with a mixed technology landscape. This article will explore ways to apply modern DevOps practices when dealing with COTS products.

COTS products are an important part of enterprise landscapes

Why should we even deal with COTS and other systems of record?

Consider the analogy of two gears.

If you can completely ditch your legacy applications, then congratulations — you don't have to deal with this and can probably stop reading this article. The rest of you who cannot do that will eventually realize that while your digital and custom applications can deliver at amazing speed now, you are still somehow constrained by your legacy applications. Speeding up the latter will help you achieve the ultimate velocity of your delivery organisation.

For example, an organisation uses a COTS product for their customer relationship management (CRM) to provide information to both a digital channel (like an iPhone app) and to their customer-service representatives (CSRs), and the speed of providing new functionality on the iPhone app is limited by the performance speed of the back-end CRM system. Increasing the delivery speed of the CRM system in this case speeds up not only the enablement of the iPhone app but also puts new functionality in front of the CSRs more quickly.

Besides limiting speed, that COTS product might also soak up a lot of effort to support several code lines at once (production maintenance, fast releases, slow releases), which has become a common pattern in organisations. The efforts required to branch and merge code and do the required quality assurance increase with each code line. I have seen code-merge activities consume up to 20% of the overall delivery effort and add weeks and sometimes months to the delivery timeline. My experience indicates that this merging effort can be reduced by up to 80%, saving millions of dollars. This figure was calculated by comparing the proportion of effort for configuration management before and after the implementation of the practices outlined in this article.

Many COTS products have not yet shifted to the DevOps world

You might wonder whether COTS vendors have understood the need to operate in a world focussed on DevOps and continuous delivery. In my experience, there is a realisation that this is important but most of the solutions provided by the vendors are not yet aligned with good practices (like bespoke SCM solutions and the lack of development-tool APIs). The guidance I offer below falls mostly in a grey area where vendors don't encourage you to use these methods (as they prefer you to use their solution) but you are not breaking anything or compromising your support arrangements. As a community, it is our responsibility to keep pushing COTS vendors to adopt technical architectures that better fit a DevOps context. In my experience, the feedback has not been strong enough yet and vendors can continue to ignore the real

Look for opportunities to make new practices easy to adopt. Good processes that are difficult to follow will hardly be followed.

needs of DevOps-focused organisations.

I would love for vendors to reach out to our community but so far I have not seen this happen. It is up to us in the industry to demand that they do the right thing or alternatively to start voting with our feet and slowly move away from their misfit solutions. When I have the choice and it is economically reasonable, I avoid introducing new applications that don't meet the following minimum requirements for DevOps:

- Can all source code, configuration, and data be extracted and stored in external version-control systems?
- Can all required steps to build, compile, deploy, and configure the application be triggered through an API (CLI if programming-language based)?
- Can all environment configuration be exposed in a file that can be manipulated programmatically?

	Number	Time for using mostly auto-merge	Time for Manual Merge
Non-Conflicting files	113	4.5 hr	90 hr
Conflicting files	2		

How to approach COTS code and get its configuration management under control

Step 1: Find the source code

COTS applications can be pretty annoying when it comes to finding the actual source code. Many of them come with their own configuration management and vendors will try to convince you that those are perfectly fine. No, not just fine — they are more appropriate for your application than industry tools. They might be right, but here's the thing: it's very unlikely that you only have that one application and it's very likely that you want to manage configurations across applications. I have yet to find a proprietary configuration-management solution that can easily integrate with other tools.

Imagine a baseline of code. You want to be able to recall/retrieve the configuration across all your applications, including source code, reference data, deployment parameters, and automation scripts. Unfortunately, this has so far not been possible for me with COTS configuration products. They also usually don't track all the required changes very well, but mainly focus on a subset of components.

Last but not least, they poorly deal with parallel development and the need for branching and merging. While I am certainly

no fan of branching and merging, more often than not it is a necessary evil that you need to deal with. In my experience, this process can be extremely costly and error-prone with COTS products and improving this alone will lead to meaningful benefits. I have seen organisations that tracked which modules were changed in a release in an Excel sheet, and their merge processes required comparing those sheets, followed by some manual activity to resolve the conflicts. Not only is this error prone, it is also quite labour intensive. By being able to store your code in a standard version-control system you can reduce the error rate to nearly zero and achieve a reduction of effort of up to 95%.

The table above shows an example of the effort required for a single merge activity

So what can you do if you don't want to use the proprietary source-control tooling? First of all, identify all the components that your application requires. The core package and its patches are better managed in an asset-management tool so I am not going to discuss those. What you want in your configuration-management tool are the moving parts that you have changed. For example, in a Siebel implementation I was dealing with, the overall solution had over 10 000 configuration files (once we exposed them) but our application only touched a couple hundred (about 2% of all files).

Storing all the other files will just bloat your configuration management with no real benefit, so try to avoid it. Especially when you later want to run a full extract and transfer, this can become a hindrance and the signal-to-noise ratio gets pretty low. If you measure percentage of code changes between releases, this is only meaningful if you analyse the code that your application changed rather than the full code of the base product.

Once you have identified all the components that require tracking, you have a few ways to deal with them:

Option A: Interfere with the IDE

The most effective and least error-prone way to do this is by integrating the developer IDE with the version-control system in the back end to intercept any changes made to the COTS application

on the fly. For example, for one of our Siebel projects we created a little custom UI in .Net that intercepted any change made through the Siebel Tools IDE, thus forcing a check-in into our version-control system with the required metadata. This UI used the temporary storage of the Siebel Tools IDE to identify the changed files and pushed them into the version-control system in a pre-defined location to avoid any misplacement of files.

(Note: When manually storing COTS config files in version control, they often end up in multiple locations because the repository folder structure does not matter to the COTS product. When importing files back into COTS products, only the file name and/or the file content is important, not where the file is located on the file system. As a result, developers will often store a (duplicate) file in a new location when they are not able to quickly

identify that the file already exists somewhere else in version control. Controlling the location of files via the mechanism described above solves this problem also.)

In Figure 1 you can see the custom IDE, which supported:

- requiring username and password for the developer to log in,
- automatically assigning a location for the file,
- allowing the developer to search for the right work item,
- allowing check-in comments, and
- providing feedback on the status of the check-in.

Option B: Extract on a regular basis

Where you cannot easily interfere with the IDE, you might want

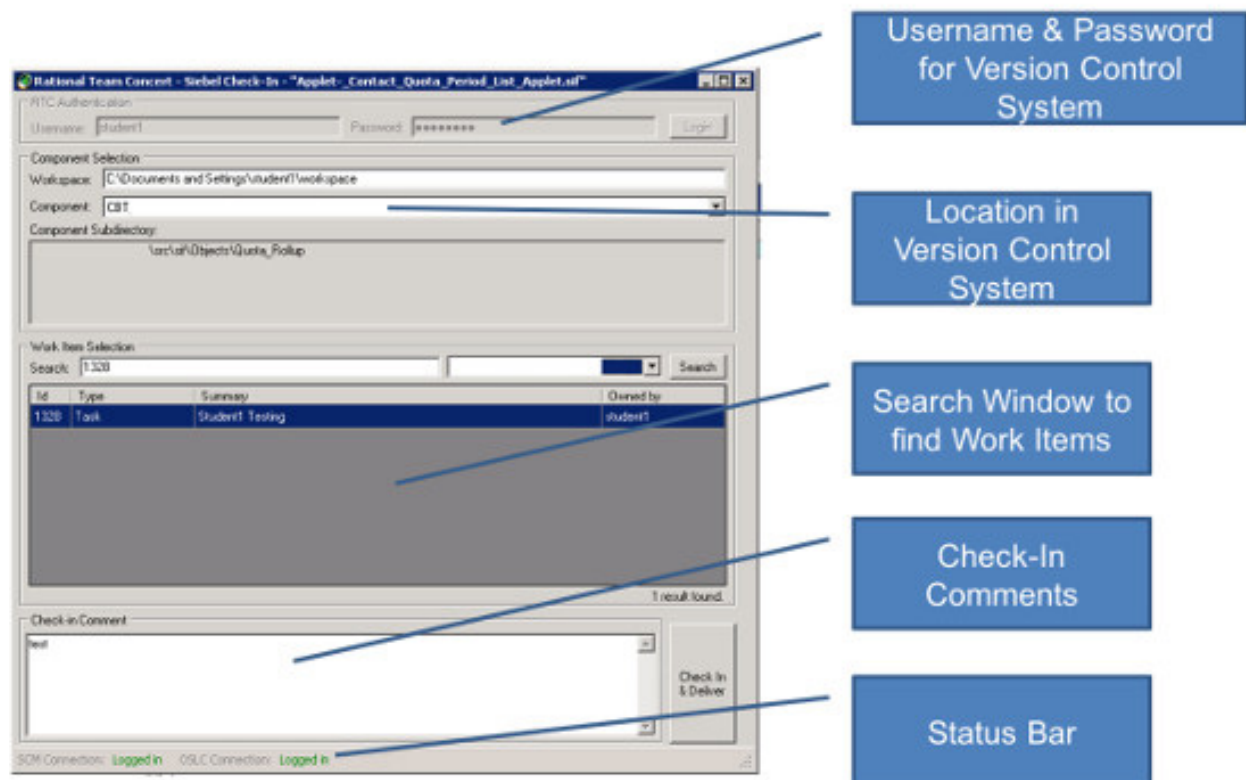


Figure 1: A custom IDE to intercept any code changes.

to use a regular extraction utility to pull configuration files out of the COTS application and push it into version control. This could be done every night or even more frequently. For the same reasons as explained in option A above, you should look for a way to identify only recent changes and not push every file into version control every time. Furthermore, while many version-control systems ignore check-ins of exact copies, the performance of this solution would be very much impacted by the number of files.

Option C: Force outside creation of files

A few years ago, I worked with some smaller COTS products that didn't allow me to follow either of the above processes as there was no programmatic hook into the UI of the IDE and it provided only import functions, not export. In this case, we changed the process for developers to basically develop outside of the COTS IDE and built automation that imported the files into the COTS product upon check-in to version control. This is clearly the least favourable

solution as it requires additional effort by the developers and increases the risk of overwriting recent changes in the environment when developers don't adhere to this process and use the COTS IDE instead. For this solution to work, we had to automate the deployment process and keep control over the environments.

Step 2: Make good practices easy for developers

Many times, developers working with COTS or legacy applications are just not used to modern development practices. Enforcing these can feel like extra overhead and can make the adoption much harder than necessary. Look for opportunities to make new practices easy to adopt.

For example, don't force mainframe developers to move their files to a different file system to check code into your preferred configuration-management system.

Don't make developers switch context to use JIRA for tracking

work items. Integrate any additional tooling into the natural steps of a developer. For example, use an IDE that can provide basic coding checks (e.g., in COBOL, to start commands from column 8) and integrate with a ticket system like JIRA.

Among mainframe developers used to developing in a text-based system, the ease of getting feedback this way may improve adoption. As mentioned, in Siebel, you can create steps in the IDE that automatically commit code into your chosen configuration-management system and make it easy to identify the work item(s) you are currently working on.

All these changes will increase developer adoption of appropriate practices — not because they are better for the team, but because they make life easier for the developers themselves. Good processes that are difficult to follow will hardly be followed.

Even obvious improvements can be hard to implement. At one company, I was trying to convince

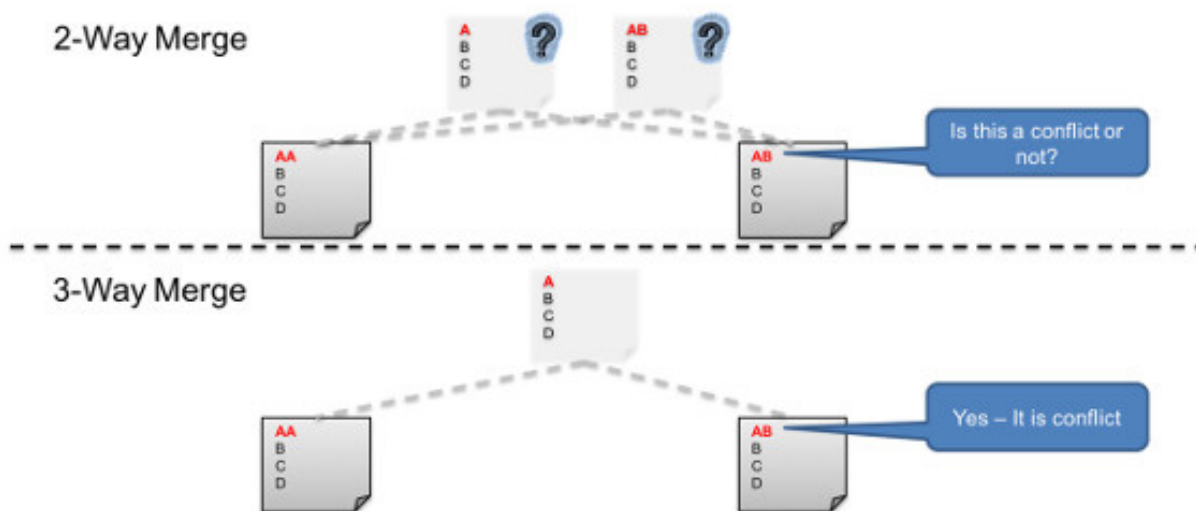


Figure 2

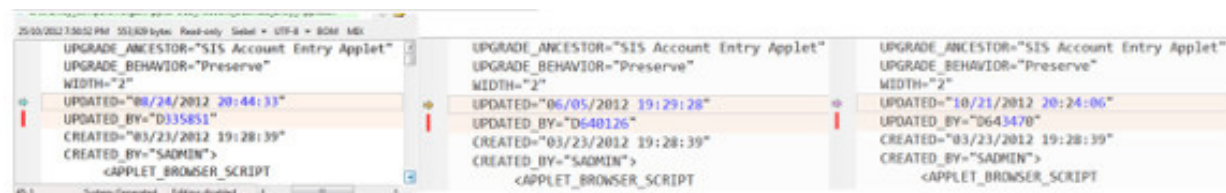
developers to use an IDE for COBOL development rather than using a text pad, as the latter would only identify basic coding issues (such as a command starting in column 7) once code had been uploaded to the mainframe. The team didn't come around to the idea of adopting the IDE until I proved that my code when uploaded failed significantly less than the average developer's code.

Step 3: Support intelligent merges

Developers who are used to native COTS products are often not familiar with [three-way merges](#). Even if they are, traditional tooling might not provide the necessary support. I will showcase this in the case of Siebel code. We will have to dive a little deeper here to explain the idiosyncrasies of Siebel code and the Siebel Tools IDE.

When you prepare to merge code natively, Siebel tools basically compare two versions of the file and show you the differences, without identifying an ancestor. You then have to judge what to do with it. Siebel tools do not know about three-way merges. Figure 2 demonstrates how a three-way merge can help identify conflicts.

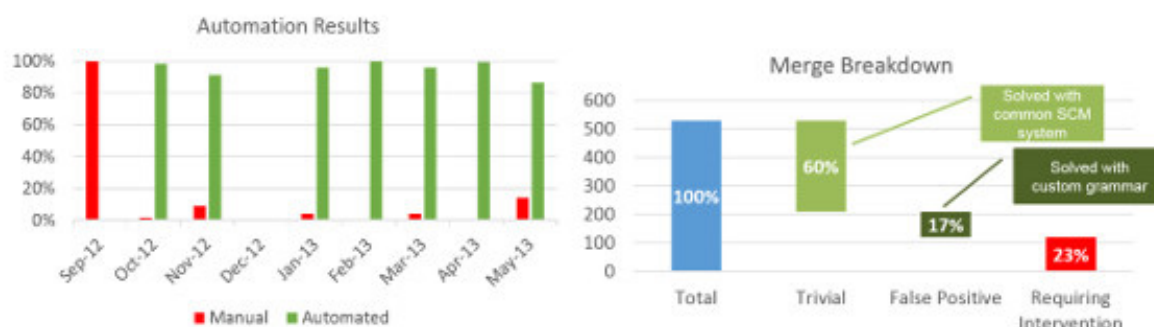
When trying to use a common configuration-management tool to enable three-way merges for Siebel, I ran into a new problem. Developers did not trust the configuration-management tool. I was surprised, but a closer look at the Siebel code showed me the problem. Let me explain this by first showing you a code sample:



As you can see, Siebel stores some metadata in the source code (e.g., user name and timestamp of change). A configuration tool that is not context aware will show you a conflict if your files differ only by timestamp but are otherwise the same. If you open the same files in the Siebel Tools IDE, which does understand that this difference is not relevant, it shows you no conflict between the files. If you run a report using traditional configuration management tools, you will see a large number of false positives.

This leaves you with the choice of [Siebel Tools](#), which avoids false positives but does not provide three-way merges, or a traditional configuration-management tool that provides three-way merges but shows a lot of false positives. Here is where better merge tools make all the difference. Tools like [Beyond Compare](#) allow you to define a custom grammar that identifies parts of the code that are not relevant for the merge. Look for such grammar for your entire COTS configuration and use the merge tool that is most appropriate.

Below are the results from my project, which show a significant reduction of merges that required manual intervention. There's also a breakdown of the different kinds of file merges.



Step 4: Close the loop by enforcing correct configuration (a.k.a. full deploys)

COTS products and other legacy systems often suffer from configuration drift as people forget to check code into version control. Because configuration management is not something COTS developers traditionally deal with, the chance is higher that someone makes a change in the environment directly without adding code to version control. This means that the application or environments do not match what is currently stored in configuration management. If something goes wrong and you need to restore from configuration management, you will miss out on those changes made directly in the environments. We want to minimise this risk and the associated rework.

The most practical way to deal with configuration drift is to redeploy the full application on a regular basis (ideally daily, but at least every week). This will over time enforce better alignment and minimise the amount of drift.

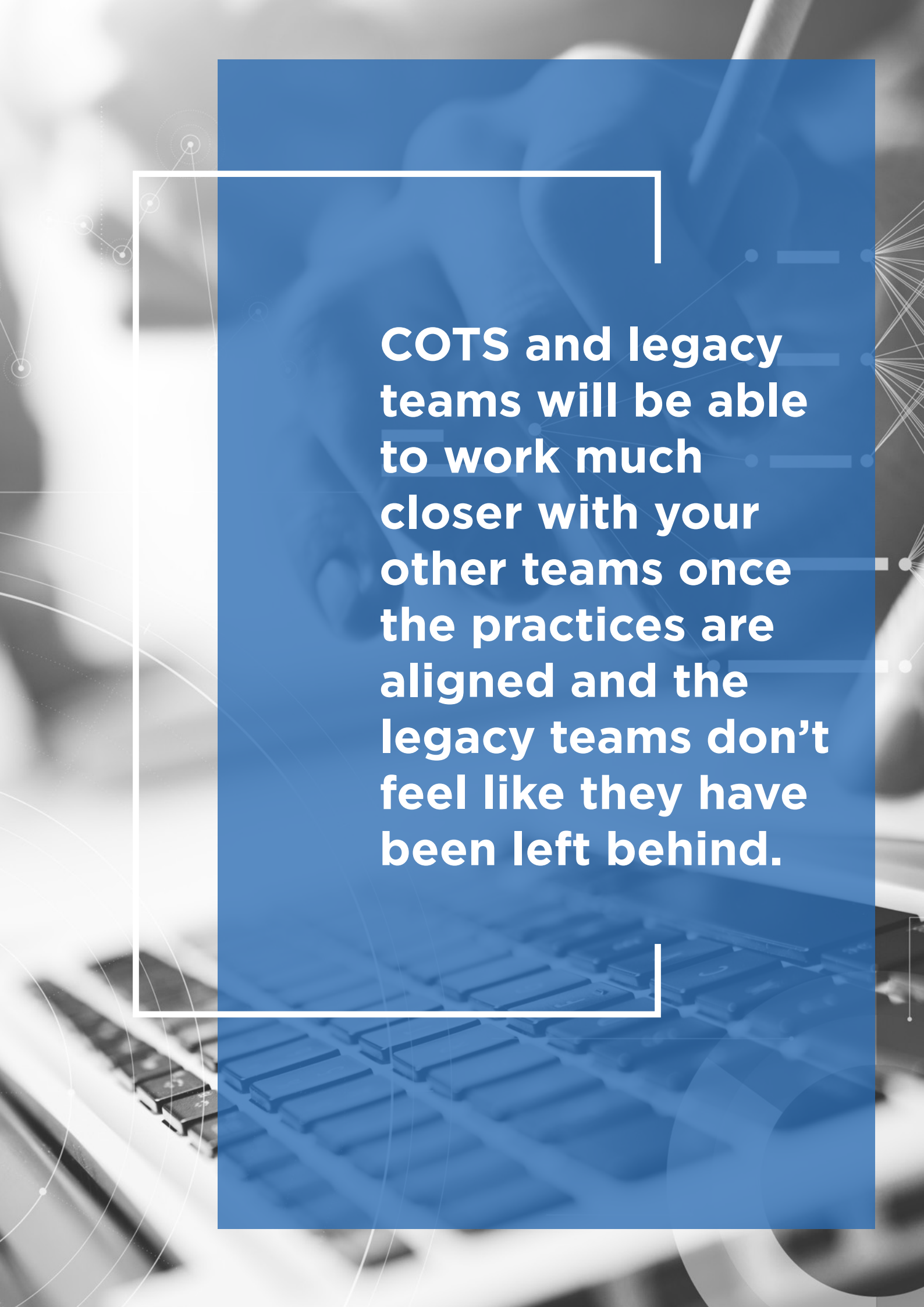
Conclusion

COTS and legacy code can behave a lot more like your normal custom code if you put some effort into it. This means that you can leverage common practices for code branching and merging, reliable environment configuration, increased resilience to disaster events, and, as a result, more predictable delivery of functionality to production.

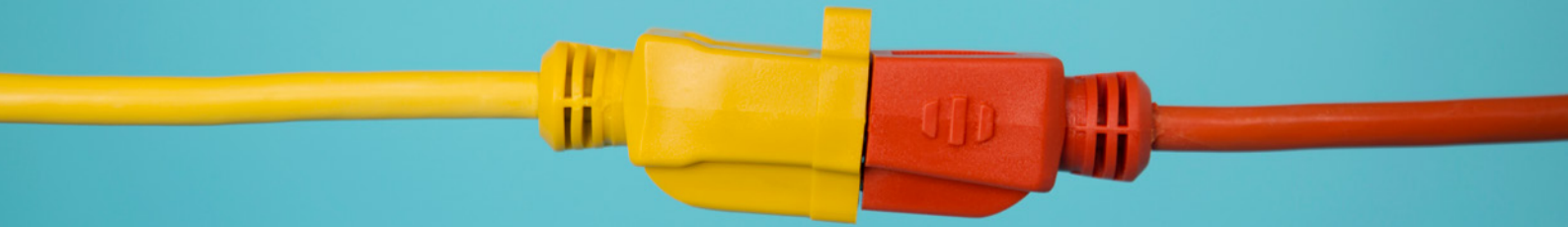
Some creativity is required and the bar is a bit higher to shift the culture in your development team but once you get there, the productivity and predictability of development will pay back significantly.

In one project, we were able to reduce non-value-added development time by over 40%. In another project, the configuration-related defects and outages were reduced by over 50%. And if that is not motivation enough, the COTS and legacy teams will be able to work much closer with your other teams once the practices are aligned and the legacy teams don't feel like they have been left behind.

You can all move forward on your DevOps journey together!



COTS and legacy teams will be able to work much closer with your other teams once the practices are aligned and the legacy teams don't feel like they have been left behind.



Read online on InfoQ

KEY TAKEAWAYS

The most popular agile framework, Scrum, predates the growth of DevOps. In consequence, the practices within Scrum (and other agile frameworks) overwhelmingly focus on what you might loosely define as the development aspects of software delivery and focus less on the operational aspects.

A blended DevOps approach requires some rethinking of teams, backlogs, how user stories are written, and so on. For example, a backlog should include scalability, deployability, monitoring, and so on.

Sprint planning should include some DevOps aspects so that we discuss not only product functionality but operability features as well.

A conventional ScrumMaster may not fit well into this blended approach — the role is more that of an agile coach.

We need to consider DevOps right from the moment we hire our team members, from the planning and building of our products through to their ultimate retirement.

MERGING AGILE AND DEVOPS

by **James Betteley** and **Matthew Skelton**

There's no point building a super-cool, super-functional product that looks and feels awesome for the customer if we can't deploy it, maintain it, and support it once it's gone live.

In the agile world, great efforts have been put into making sure we deliver what the customer expects, within reasonable budget and on time. We also go to great lengths to help our customers determine the highest-priority features, so that we shift our focus towards delivering high business value. We deliver early and often to get regular and relevant

feedback. We use user stories to help us think from an end user's perspective, and we test our code on every commit to make sure we're not breaking our codebase.

This is great, but where are all the clever tricks and techniques designed to ensure we deliver deployable, scalable, high-performing products that we can update in real time, monitor from the very second they're built, and manage from day to day without needing a team of support engineers?

Agile has borrowed (and continued to evolve) great ideas from the automotive industry, neuroscience, ancient philosophy, the military, and mathematics, to name but a few (think lean manufacturing, cognitive bias, servant leadership, planning, and relative sizing). It's now time to borrow some thinking from the DevOps scene to ensure that agile remains the most suitable and successful set of principles and practices for delivering products.

Most products spend the majority of their lives being supported and maintained after they've been launched (bug fixes, feature releases, and enhancements, for example). The practical way in which we manage these (rolling out changes to a live service, testing in live-like environments, and so on) and how the product can scale for performance are seen as operational features, and are often nowhere to be found on the product backlog.

A [2014 report](#) in ZDNet cites a survey from consulting firm CEB, which "found that 57% of the budget will go towards maintenance and mandatory compliance activities, down from 63% back in 2011." A [Gartner report](#) from 2006 put the figure as high as 80%.

DevOps teaches us that operability (that is, operational features) is actually a first-class citizen, and should be treated with as much regard as any other product feature. The best way to ensure this happens is to foster a strong culture of collaboration between development teams and operations. How we achieve this collaboration is another question, and DevOps models can differ quite wildly, from the Amazon "You build it, you run it" approach, where both development and operational activities exist within a single product team, to the "DevOps as a platform" approach found in some Google teams.

The need for DevOps

Agile and DevOps have lived side by side for a few years, and there's been plenty of discussion around the relationship between the two.

Some people see DevOps as a subset of agile, others see DevOps as "agile done right", and others see DevOps as a set of automation practices, loosely connected to the agile big picture. It all depends on our individual definition of DevOps. Regardless of how we see DevOps, the intention of delivering working software that can be managed, maintained, scaled, supported, and updated with ease is something the software-delivery world desperately needed.

The way we run and operate software has changed massively since agile frameworks were invented. Scrum started back in 1993, [DSDM](#) launched in 1994, and the XP book was launched in 1999. Back then, we were writing MSI installers, burning them to disks, and posting them to people!

Running, maintaining, and operating software was generally not

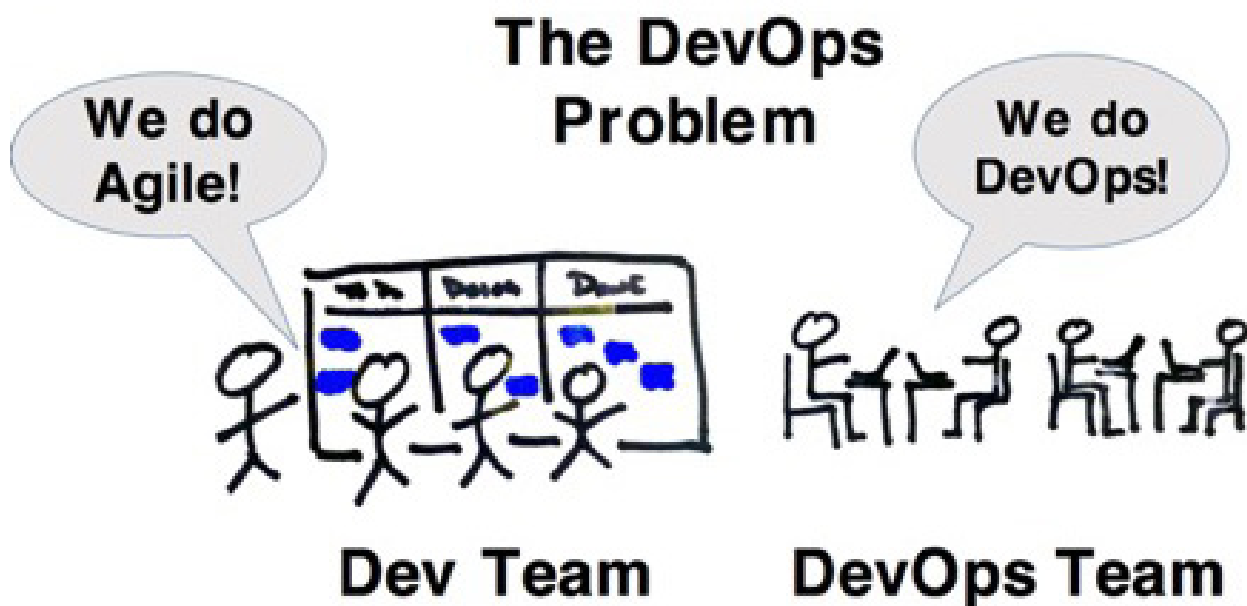


Figure 1

something most software developers were involved with in any way.

Since then, a major shift to SaaS and PaaS has occurred and the production environment is at our fingertips. Developers are now actively involved in the operation and support of their systems, but we're still following frameworks that don't accommodate this change in the way we work.

Continuous delivery to the rescue, almost

Continuous delivery requires deployment automation. This was a step in the right direction — even if it did, in some organisations, inadvertently create a spin-off profession of continuous-delivery engineers (thus often creating another silo). Continuous-delivery engineers grew into continuous-delivery teams and, eventually, platform teams as infrastructure management became increasingly central. In many cases, this shifting out of the continuous-delivery aspect into a separate team seemed

natural, and allowed many agile teams to get back to what they felt most comfortable with: developing software rather than delivering it.

Unfortunately, this fits the Scrum framework quite nicely — the development team focuses on designing, developing, and testing their software and the continuous-delivery team focuses on managing the system that deploys it and the underlying infrastructure.

The trouble with this approach, of course, is that by separating the build and deployment-automation along with the infrastructure management, we're essentially making it somebody else's problem from the perspective of the agile team, and operability once again disappears into the background.

Many teams did embrace continuous delivery the "right way" and that enabled them to adopt a "we build it, we run it" approach to software delivery (and with that a greater sense of ownership

and improved quality as a result), but the same cannot be said for everyone. Evidently, there's considerable resistance to adopting new practices into particular agile frameworks, even if those practices are themselves agile to the core. And now we're seeing the same thing with DevOps.

DevOps anti-patterns

The main focus of DevOps is to bridge the gap between development and ops, reducing painful handovers and increasing collaboration, so that things like deployability, scalability, monitoring, and support aren't simply treated as afterthoughts.

However, we've already started to see strong anti-patterns emerging on the DevOps scene, such as the separation between the development team and the DevOps team, effectively creating another silo and doing little to increase collaboration. (Figure 1)

The problem is that there is very little information from a practical perspective on how to actually

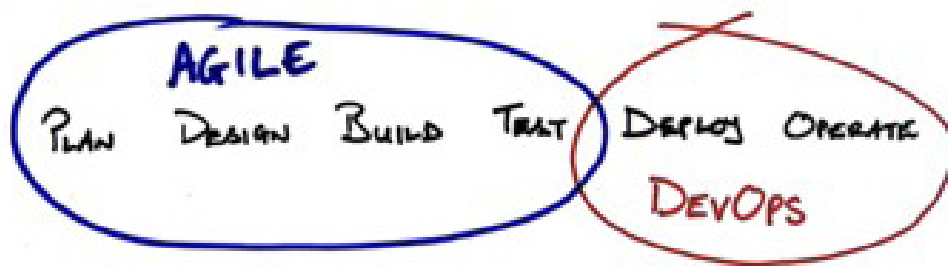


Figure 2

blend your agile development teams with this new DevOps approach.

What practices do we need to adopt? Which practices do we need to stop? How do we get started? What roles should we have in the team? These questions remain largely unanswered. As a result, teams are bolting on DevOps rather than fully integrating it into their software development processes. (Figure 2)

In this classic DevOps anti-pattern, we have all the agile ceremonies happening, and many of the usual DevOps practices as well, but the end result is no better than before — operability is still an afterthought and products are optimised for development rather than for delivery and operation. This is all because the key DevOps practices are being bolted on rather than being present from the start.

The solution, of course, is to bake these good DevOps practices in from the very beginning, by absorbing them in to our daily agile processes and practices — and this is what requires some tweaks to our agile frameworks.

Updating agile practices

So what can we do to ensure we're developing software in an agile manner, while also delivering and maintaining our products and services in accordance with

some of the latest and greatest DevOps best practices? Well, it's easy — we just shift left!

That sounds a lot easier than it is in practice, but the concept is straightforward enough. We underline this by adding operability tasks/stories to the backlog, alongside our user stories. Our backlog suddenly becomes a full set of epics, stories, and tasks needed to get our product delivered successfully and then maintained once it's gone live (as opposed to simply a set of functional features from an end user's perspective).

On the surface this might sound easy, but there are a couple of considerations:

- Who's going to work on these operability stories/tasks?
- How can we write an operability story if there's no end user?
- What are these so-called DevOps best practices?
- How can a product owner be expected to manage this?

The answer to all of these questions is: "Things will need to change."

Teams

Most agile teams we work with don't include ops, support, or infrastructure specialists. We might argue that there's insufficient demand for such specialisms to be

in each and every agile team, and we might be right, but don't forget people said that exact same thing about testers, and architects, and database engineers, and UX, and so on....

If the way we deliver, support, update, scale, and maintain our product is important, then we need these skills in our team.

Is this going to mean that you have to break Jeff Bezos's "two-pizza team" rule? Maybe. But if our share of pizza is that important, we could always skill up! (This isn't actually as daunting as it sounds — the more we move towards an X-as-a-service world, the less hard-core sysadmin knowledge we need. Instead, we'll all need a firm understanding of cloud functions and related services.)

Backlogs

If we have cross-functional teams, then we're going to need cross-functional backlogs.

Leave the traditional view of a product backlog in the past — it's time for a fresh approach that embraces the operability aspects of our services. And we use the term "services" intentionally, because what we tend to build these days are indeed services, not shrink-wrapped products. Services are products that need to be deployed, scaled, maintained, mon-

We need to consider
DevOps right from
the moment we hire
our team members,
through the planning
and building of
our products right
through to their
ultimate retirement.

itored, and supported, and our backlog needs to reflect this.

Most Scrum product backlogs that we see contain something like 90% traditional features that can best be described as a collection of desirable features from an end-user's perspective. The remaining 10% tend to be performance related or something to do with preparation (setting up dev environments, prepping databases, and so on). The weighting towards end-user functionality/product features is revealing. This could be a consequence of the Scrum framework itself or a result of end-user bias by product owners (or something else entirely).

Instead, a modern service backlog should describe (besides user functionality):

- the scalability of the product/service (up, down, in, out — and when);
- the deployability (Does this need to be deployed live with no down time?);
- Monitoring of the service (What aspects need monitoring? How do we update our monitoring with each change?);
- logging (What information should be logged? Why? And in what style?);
- alerting (Who? When? How? Why?);
- the testability of the service;
- security and compliance aspects such as encryption models, data protection, PCI compliance, data legislation, etc.; and
- operational performance.

As one of us (Matthew Skelton) has written: To avoid “building in legacy” from the start, we need

to spend a good portion of the product budget (and team time) on operational aspects. As a rule of thumb, [I have found that spending around 30% of product budget on operational aspects produces good results](#), leading to maintainable, deployable, diagnosable systems that continue to work for many years. It should be noted that these operability and security requirements are continually changing, and evolve with the product/service, so one cannot simply get them all done at the start of the release and then move on to the traditional product features. For example, it may not be cost-effective to implement an auto-scaling solution for our system until that system is commercially successful. Or perhaps we need to change our encryption model to conform to a new security compliance. Equally, we may need to change our very deployment model when new geographic locations come online. Also, monitoring will usually need updating when any sizeable change to an application's functionality takes place.

User stories

User stories are a fantastic way to capture product requirements from the perspective of the expected outcome. User stories have helped many developers (ourselves included) to think about problems from the end user's point of view and to focus on solutions to problems rather than simply following instructions. We're referring, of course, to the way that user stories focus on the “what” rather than the “how” (a good user story presents the problem and leaves the solution up to the developers).

User stories are often written in this format:

“As a ___,

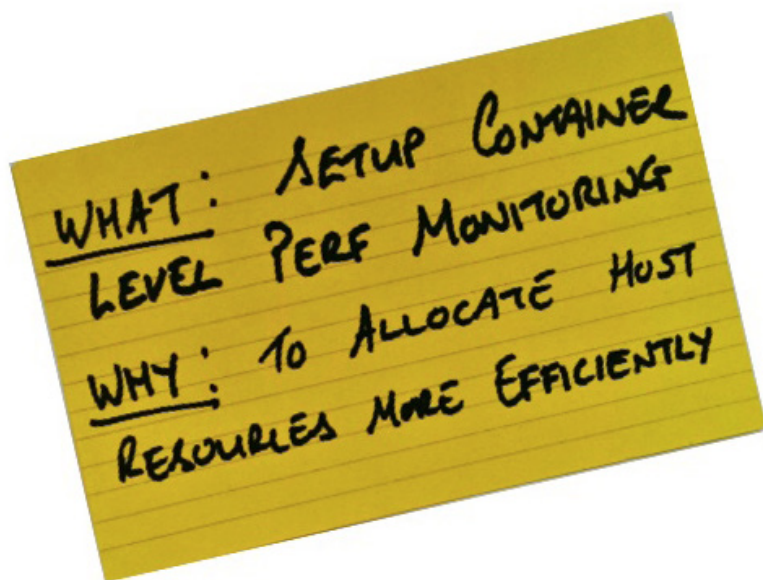


Figure 3

I want___s

o that___."

This forces us to write them from a user's perspective (although not necessarily an end user).

However, over the years, we have found that writing operability stories using this format doesn't really offer the same benefit. This may be because the user perspective has no impact on the way the solution is technically implemented. Regardless, it feels a little redundant writing "As a sysadmin" or "As a developer" if we're implementing the solution ourselves.

It's not particularly unusual to see technical backlog items written in a backlog without adopting the "As a___, I want___ so that___" format, and similarly we tend not to recommend the format for operability features. We instead prefer to use a "what and why" format, which simply lists what needs doing and why (to provide context). (Figure 3)

Sprints

Two-week sprints feel about right for developing new features, testing and deploying, and then demonstrating them to stakeholders. Any longer and it would be hard to maintain focus, as well as pushing out the feedback loop to a slightly uncomfortable delay. Any shorter — say, one week — and suddenly meetings and other ceremonies take up an inordinate percentage of our sprint time, meaning the amount of work we can get done feels tiny. So two weeks feels right for many people. It's just the right amount of time to get your head down and focus on what you're committed to doing.

This is great if you're developing a new product, but what if you're iterating through some improvements or developing the next version of your product?

Who's going to look after all the constant issues that arise from the production platform?

If we're exposed to frequent interruptions such as these (or equally damagingly, varying degrees of such interruptions) then we're

well aware that they can wreak havoc with our sprint commitments. Two-week sprints seem to add a lot of value in terms of helping people to focus on a realistic target, but an unpredictable environment can make it hard to determine exactly how much of our backlog we can achieve — difficult, but not impossible.

If we measure the average amount of work we can burn through from our backlog, and the average amount of interruptions we get from the production platform, we can essentially deduce two velocities.

Our backlog velocity is the rate at which we can complete planned work from the product/service backlog while the unplanned velocity is the capacity to handle work that hits the team during the sprint. Tracking these two velocities allows us to plan more effectively.

Kanban is of course another option, which can accommodate both planned and unplanned work, and is often the framework of choice for teams who have little insight into what they'll be working on in a week's time. It can also be highly effectively to deliver longer-term projects/releases, but requires high levels of discipline in ensuring that the backlog is continually and correctly prioritised.

Sprint Planning

If we're doing sprints, then we'll need to do sprint planning. To bring a DevOps perspective to our sprint planning, we need to do the following:

- Invite ops/infrastructure/support people to the planning session.

We have to take a fresh look at some well-established concepts within Agile, such as the skillsets and roles within a Product Team, the Product Backlog itself, and how we plan and execute iterations.

- Discuss not just product functionality, but operability features as well.
- Plan their place in the upcoming sprint.
- Take into consideration time and effort that will be consumed by interruptions — that is, unplanned work coming from the production platform such as bug fixes, escalations, etc. This value is our unplanned velocity and effectively acts to reduce our backlog velocity. The higher our unplanned velocity, the lower our backlog velocity will be.

Definition of done

A popular definition of done is “passed UAT”, which is basically another way of saying “the business has signed off the feature”. But this largely forgets about operability, security, performance, and so on. For a story to be considered done, it needs to be ready to go live (or better yet, be in the live environment already). This means it needs to be scalable, functional, monitored, secure, and obviously deployable! If our story doesn’t satisfy all of these, it’s not done.

ScrumMaster

Bearing in mind that we’re going to need to bend or break some existing Scrum rules (see above for examples), the role of the ScrumMaster is thrown into question. Even if we want to maintain a process that closely resembles Scrum, the fact is it isn’t Scrum; it’s going to be a blend of Scrum and DevOps.

Certain aspects of the ScrumMaster role are still perfectly valid, such as removing impediments, but the ScrumMaster will now need to remove impediments not just to software development

but also to software delivery and maintenance.

Another option is to transition the role to an agile coach, who adheres to the principles and values of agile in a way that’s sympathetic to our new processes and not constrained by the prescriptive rules of the Scrum framework. The last thing we want is a ScrumMaster who doesn’t appreciate the purpose of DevOps; that’s simply going to create an even bigger divide between the development and operations sides.

Product owner

In our blended agile and DevOps environment, our product owner needs to understand the importance of operability more than anyone.

In SaaS, PaaS, and serverless environments, a lot of the value is hidden — it’s not in the front end. The value is in how our services work. It could result in saving time, saving money, increased performance, reduced risk, improved reliability, or any other hidden value. Product owners now need to get this, because ultimately they’re responsible for guiding the priorities.

Continuous integration (CI) and continuous delivery (CD)

Some people recommend separating CI and CD tooling, presumably because while CI is more dev-focused, CD has a more holistic view.

Whichever way we look at it, CI and CD are more than just tools; they’re actual ways of working. There are big differences between having a CI system and doing continuous integration. The same can be said of CD.

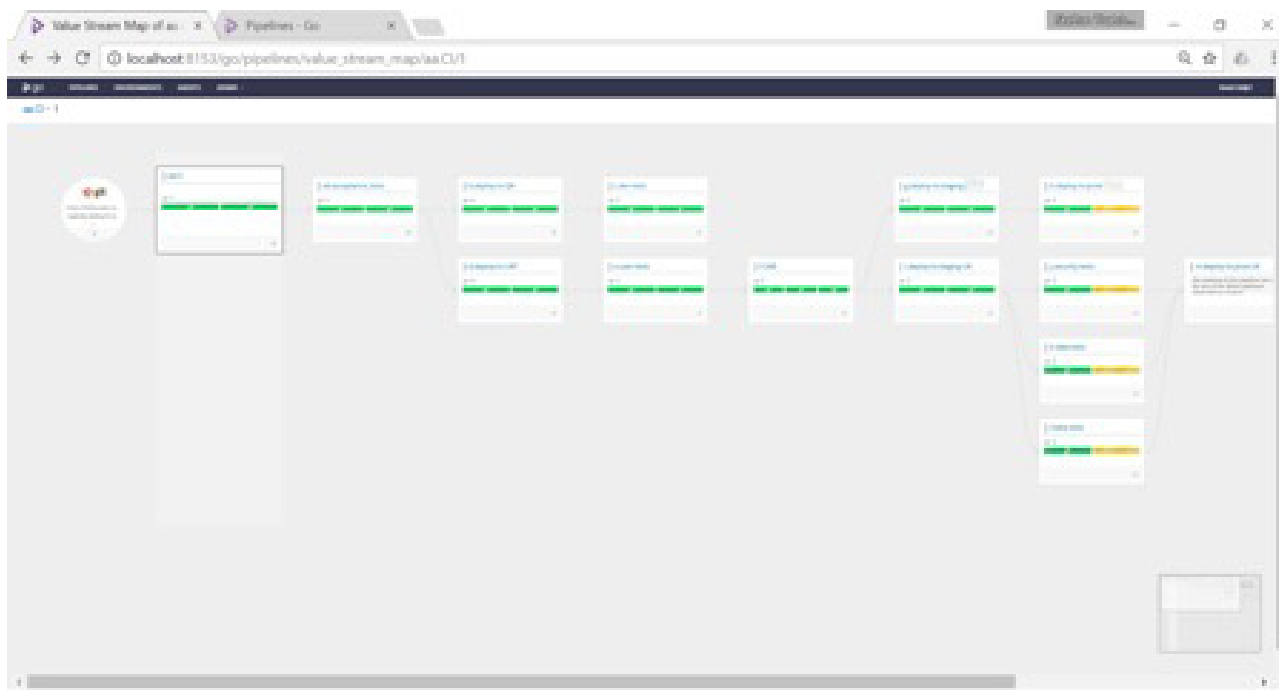


Figure 4

In our DevOps/agile blended environment, it's essential that we not only use CD as a delivery mechanism but also as a guiding set of principles and practices. This matters because CD brings development and operations into the same frame. A good CD pipeline like the one below will visualise all of the important steps in getting software delivered successfully and regularly — we can see how important it is to have available test infrastructure, reliable testing frameworks, good monitoring, and deployment automation. (Figure 4)

Remember the [eight principles and four practices of CD](#), as outlined by Dave Farley, paying particular attention to the key practices of “build binaries only once”, “use precisely the same mechanism to deploy to every environment”, and “if anything fails, stop the line!” — but above all, heed this message: “Everybody has responsibility for the release process.”

Conclusion

The most popular agile framework, Scrum, was designed for a time when teams tended not to worry about operational issues such as scalability, deployability, monitoring, and maintenance.

As a result, practices within Scrum (and other agile frameworks) overwhelmingly focus on what we might loosely define as the development aspects of software delivery, and less focused on the operational aspects.

DevOps helps to redress that imbalance, but has little influence over the practices that happen during the development phase itself. The lack of definition around DevOps and the lack of a prescriptive framework mean that there's little or no information on how to bring DevOps thinking into our agile software-development processes.

To maximise the value of agile and DevOps, we must start to implement some of the DevOps

principles right at the beginning of our development process, because bolting on a bit of deployment automation at the end isn't going to help us build more scalable, deployable, and manageable solutions.

We need to consider DevOps right from the moment we hire our team members, through the planning and building of our products, right through to their ultimate retirement.

This means we have to take a fresh look at some well-established concepts within agile, such as the skillsets and roles within a product team, the product backlog itself, and how we plan and execute iterations.

Many teams have successfully adapted their agile practices to become more DevOps aligned, but there's no one-size-fits-all solution available, just a collection of good patterns.

Read online on InfoQ



DESIGNING DELIVERY

BOOK REVIEW AND INTERVIEW

by Manuel Pais

Jeff Sussna's book, *Designing Delivery*, not only dives into how organizations should think about their business approaches but also shows deep understanding of the expectations of customers today. These include speed of delivery of new features (or bug fixes), operational excellence (access anytime, anywhere) and active brand engagement (across multiple platforms, device) during the entire customer lifecycle.

O'REILLY®

Designing Delivery

Rethinking IT in
the Digital Service
Economy



Jeff Sussna

In short, the book puts today's challenges as a service company (and most businesses have become one by now) into a coherent narrative and a comprehensive structure of capabilities required to succeed. Although it clearly targets an audience of C-level executives (as they should be able to see or have access to their organizations' global picture and its shortcomings), the executives should be wary that attempting to become a "customer-centric brand" that continuously redesigns itself requires pre-existing levels of maturity. If development teams are not agile, operations don't embrace DevOps, or design teams don't value design thinking, then they should start there.

For everyone else, this book is an interesting read and a prediction of the kinds of skills and attitudes most service organizations will come to demand from their people.

Part I of the book explains why service delivery must be a customer-centric process: never finished, continuously adapting to customer needs. A paradigm shift is required for roadmap-driven organizations that optimize internal delivery and management processes. They instead need to focus on seeing their product value from their customers' perspectives, and realize how important it is to listen and act swiftly on customer feedback.

Part II breaks down the redefinition of quality under this new business modus operandi. It frames the expectations of customers (often unaware themselves) in four dimensions: customer outcomes (jobs to be done), access (anytime and anywhere needed), coherency (across time and multiple customer touch points), and continuity (adaptation to evolving customer needs). QA's new role needs to include not only validation of the customer journey but also the internal journeys of the other roles involved in service design such as development, operations, customer support, etc.:

"QA is about validating a service's ability to harness change and helping them continuously improve."

Part III introduces promise theory as a basis for thinking of services as chains of promises made to customers. Inherent in a promise is the possibility of failing to fulfil it. Thus, a service organization thinking in promises is preparing and embracing failure, as long as it leads to active learning which in turn can lead to more successful promises. Tightly linked to this view is the understanding that all socio-technical systems we use today are complex in nature, and thus fail in unpredictable ways. Sussna writes:

"Asking whether a service is making the right promises naturally maps to validating requirements. Asking what a service needs to do to keep its promises naturally maps to identifying implementation holes and bugs. Asking what other promises a service needs naturally maps to integration testing."

InfoQ reached out to Sussna in order to better understand some of the ideas behind this book.

InfoQ: What was your motivation to write this book?

Jeff Sussna: I'd been giving a series of talks at various conferences over the course of a couple of years. All my talks were essentially about the same thing: the need to integrate design with engineering in today's IT organizations. During the same period, I read a biography of Norbert Wiener and immersed myself in the history and theory of cybernetics. I also encountered Mark Burgess's work on promise theory, which is very cybernetic at heart. These themes all came to-

gether in my mind; the book was an attempt to coherently communicate a new way of thinking along with a method for applying that new thinking.

InfoQ: How did your professional experience influence its content?

Sussna: I've built systems and led organizations across the entire development/QA/operations spectrum. I've worked on everything from compilers to content-management systems

to data-center automation platforms. That background gives me a somewhat unique perspective on IT in general and DevOps in particular.

I also have a liberal arts background. I grew up looking at pictures of Frank Lloyd Wright buildings; in college, I studied anthropology, sociology, and art theory. A few years ago, I had an epiphany when I read Tim Brown's book on design thinking, *Change By Design*. I suddenly understood that while I was not a designer in the formal,

traditional sense, I approached problems from a design-thinking perspective. From there, I discovered the design-thinking offshoot called service design. It occurred to me that, since IT is more and more about service and more and more directly relevant to ordinary people, we need to design it as we would any other service.

In sum, that's a long-winded way of saying that my background, like the content of the book, is very interdisciplinary.

InfoQ: You mention that cybernetics is a concept that precedes the sci-fi image most of us have of it. Would you like to explain why the original meaning is important?

Sussna: At its heart, cybernetics views control as circular rather than linear. Even at the simplest level, does the thermostat control the temperature of the air in the room or does the air control the thermostat? The answer is "yes." As we move from a product economy, which stressed pushing things and messages towards customers, to a service economy, which stresses companies and customers co-creating value together, we need to take a more circular approach to control. Also, as our world and the problems we face become more complex, we need to think more in terms of steering our way through complexity rather than engineering static solutions. The word "cybernetics" comes from the Greek *kybernetes*, which literally means "good steering". Problem solving in the face of complexity is less about "Does it work?" and more about "Are we steering well?"

InfoQ: The book stresses that today's organizations are de-

finied by how well and quickly they adapt the services they provide to ever-changing customer needs. Is it fair to say then that service delivery must encompass multiple areas, not only the technical delivery (IT) area?

Sussna: Absolutely! Customers judge service value by the entirety of their relationship with the provider. Enjoying a restaurant requires not just good food and not even just good food and service, but also good ambiance, price, parking availability, surrounding neighborhood, and a decent website for looking up the menu. As all business becomes service business, and all business becomes digital business, the boundaries between IT and the rest of the organization begin to dissolve. If your functionality is great but it doesn't scale, or isn't secure, or is hard to on-board, or your customer support is poor, or the website that explains what your service actually does and how/why to use it is incomprehensible, or you handle outages or security breaches clumsily... — any and all of those defects can degrade quality in your customers' eyes.

InfoQ: What are the effects, at the organizational level, of this idea of continuously evolving services? Do they require new organizational structures to be successful?

Sussna: They definitely need people and groups to work across disciplines in new ways. Personally, I'm less concerned with explicit org charts than with behavior and attitude. I think it's perfectly fine for designers and developers and testers to report to different managers. What they do need to do is to collaborate and empathize with each other, and to see

themselves as providing service to each other, on an ongoing basis, with a greater shared value (serving the customer) in mind.

InfoQ: Your ideal cocktail for continuous evolution of digital services includes not only cybernetics, agile, and DevOps but also design thinking. However, it seems that design work is still disconnected from the technical delivery work. Do you agree and how would you improve those connections?

Sussna: Integrating design with engineering isn't just about getting designers and developers to work more closely together. It's also about applying design to IT itself, and approaching IT as a design act. How do we reimagine, for example, ITIL incident management, and helpdesk software and workflows, in user-centered, service-centric ways? Do we focus on closing tickets or on helping users solve their problems?

InfoQ: Could you briefly explain what continuous design is for you and how it fits with continuous delivery and quality?

Sussna: Continuous design breaks down the boundaries between design and operations. Just as the thermostat continuously responds to changing air temperature, so too the continuous-design organization continuously responds to feedback from operations. Continuous design operates on multiple levels:

1. Continuous through time — unlike designing a coffee mug that you hope to produce and sell for 40 years, the design process never ends.

2. Continuous throughout the lifecycle — A/B testing, game days, and chaos monkeys imply that design happens in production as much as anywhere else.
3. Continuous throughout the organization — each part of the organization, whether the design or development department, or the two-pizza microservice team, is a service provider to other parts of the organization, and thus must continuously design and deliver its capabilities on its consumers' behalf.
4. Continuous delivery is a necessary but not sufficient component of continuous design. You could say that continuous quality is a measure of the success with which an organization is continuously designing.

InfoQ: The quality of a finished software product used to be characterized by the number of defects found and, in business terms, the number of sales or licenses. How does the definition of quality change when thinking in terms of continuously running services?

Sussna: The reason we need to approach digital business cybernetically is that we are actually continuously failing. Even if we design and implement a new feature perfectly, as soon as the customer starts using it, their needs begin to change. "This is great; now can you make it do X?" is a common customer refrain. The very act of using something changes the user, thus changing the design problem. We are therefore continuously narrowing and then discovering/creating new gaps between needs and capabilities. For this reason, quality has

to shift from stability, expressed as mean time between failures, to resiliency/adaptability, or mean time to repair. We are succeeding as long as we are listening and responding to our customers.


InfoQ: You mention every organization is now part of a service ecosystem, where everyone plays both producer and consumer roles. What are some good practices for managing your service dependencies?

Sussna: This is precisely where promise theory comes in. The use of the word "promise" means first of all that we make commitments, and work hard to honor them, regardless of the promises made to us. On the other hand, it means that we don't assume the promises we rely on will be kept. Whether we're talking about human promises ("I promise to help you solve your problem when you call customer support") or system promises ("I promise to serve webpages in < 10 ms, regardless of the number of users"), delivering service as promises creates the loose coupling combined with attractive force that lets coherent systems emerge without causing large-system brittleness.

InfoQ: Do you believe customer-centric brands require a new approach to service support in order to re-engage disgruntled customers and channel their feedback to the rest of the organization?

Sussna: I believe the entire service organization, from customer support back, needs to shift their mindset from "making things for customers" to "helping customers achieve desirable outcomes". Perhaps ironically, ITIL captures this definition of service perfectly. If we make that shift, we create an

attitude and set of behaviors that lets that feedback flow through the organization, instead of being blocked by organizational and behavioral silos.

A person with a beard and glasses is looking at a laptop screen. The image is partially obscured by a large blue rectangular overlay. Inside the blue overlay, there is a white rectangular frame. The text is centered within this frame.

**As our world and
the problems we
face become more
complex, we need
to think more in
terms of steering
our way through
complexity rather
than engineering
static solutions.**

PREVIOUS ISSUES



Cloud Native

In this eMag, the InfoQ team pulled together stories that best help you understand this cloud-native revolution, and what it takes to jump in. It features interviews with industry experts, and articles on key topics like migration, data, and security.



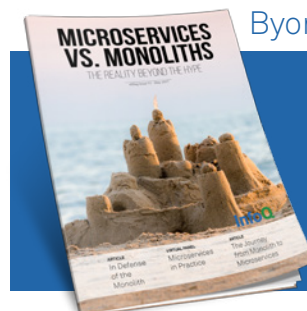
Reactive JavaScript

This eMag is meant to give an easy-going, yet varied introduction to reactive programming with JavaScript. Modern web frameworks and numerous libraries have all embraced reactive programming. The rise in immutability and functional reactive programming have added to the discussion. It's important for modern JavaScript developers to know what's going on, even if they're not using it themselves.



Serverless Computing

In this InfoQ eMag, we curated some of the best serverless content into a single asset to give you a relevant, pragmatic look at this emerging space.



Microservices vs. Monoliths - The Reality Beyond the Hype

This eMag includes articles written by experts who have implemented successful, maintainable systems across both microservices and monoliths.