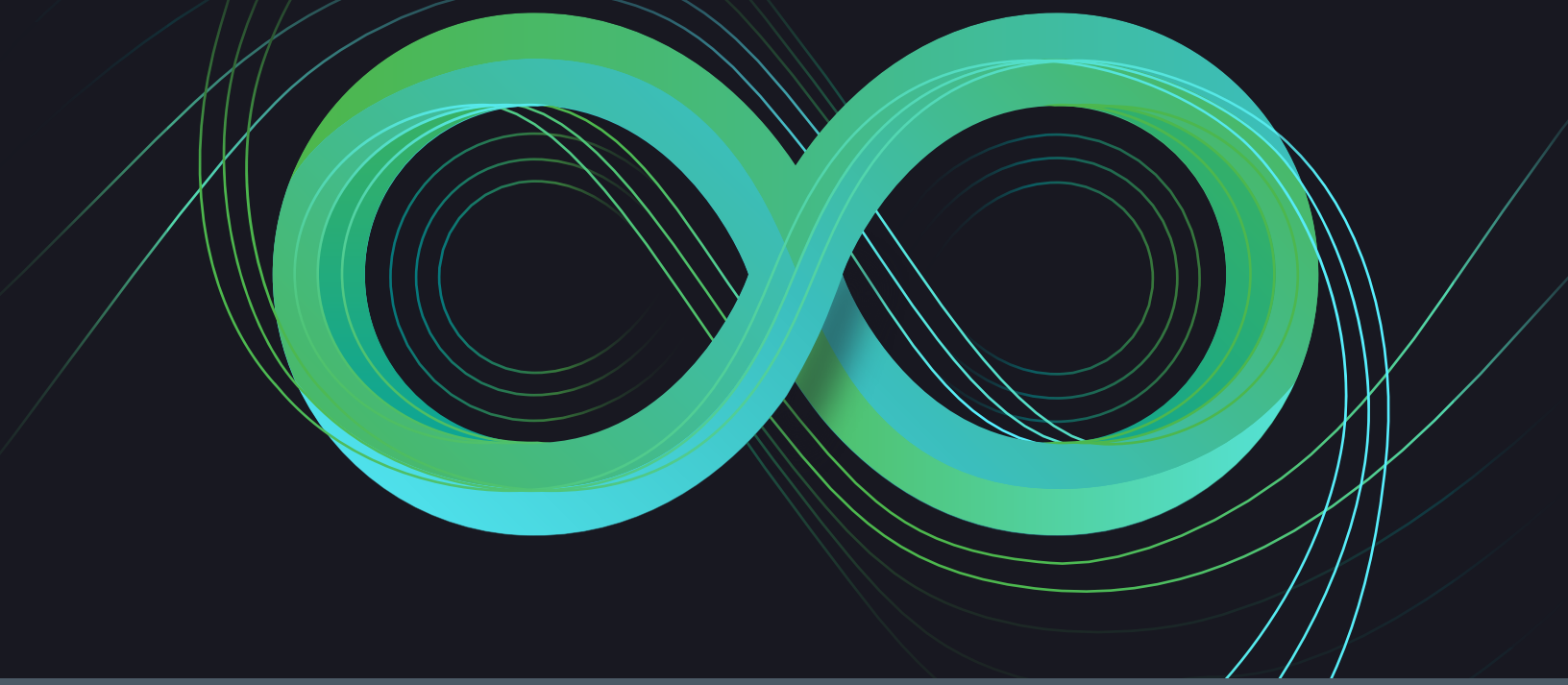
DZone

THE DZONE GUIDE TO

DevOps

Continuous Delivery and Automation

VOLUME IV



BROUGHT TO YOU IN PARTNERSHIP WITH

Automic™ bmc ca technologies cloudbees

Electric Cloud SAUCELABS Travis CI weaveworks

DEAR READER,

Ever since the publication of *Continuous Delivery and The Phoenix Project*, there's been a noticeable surge of hype around DevOps in the development world, prompting many to automate their deployments and improve collaboration between development and operations. Since then, startups offering a new automation, pipeline, communication, or monitoring tool have risen, fallen, been bought out, or become juggernauts in a very competitive field, all while educating developers on the benefits of Continuous Delivery and introducing a whole language's worth of jargon to the world.

As we take our first steps into 2017, we can see that, finally, the benefits of Continuous Delivery have really taken root in the employees and managers of today's business. Our audience has reported that more projects have implemented Continuous Delivery, more organizations have dedicated DevOps teams, and barriers to adopting these practices are being overcome. It's evident that we're moving to a more clear sense of understanding around DevOps.

However, the understanding around how to use technology to reach these goals is not quite there. For example, our analysis found that teams using container technology, such as Docker, experienced a 20% faster mean time to recovery than teams that do not. Yet only 25% of DZone members are using containers, while another 25% are still evaluating the technology. Microservices in particular have started coming into their own. The benefits of modular architecture to quickly push or roll back changes to an application are in clear alignment with Continuous Delivery, and audience members using microservices recover from failure after 7 hours on average instead of 29 for those who don't.

We can see a thirst for specific knowledge and case studies across over two dozen planned DevOps Days in 2017, the DevOps Enterprise Summit (organized by Phoenix Project author Gene Kim), future Velocity conferences from O'Reilly, product-specific conferences, and of course from DZone readers like you who visit our DevOps zone every day.

Of course, understanding is only the first step, but actually implementing Continuous Delivery across the entire enterprise is another undertaking altogether, and we've worked to address these issues in DZone's *Guide to DevOps: Continuous Delivery and Automation*, from the benefits of automation, to the pitfalls of delayed branch merges, and the folly of "one-stop-shop" DevOps solutions. Most will agree that the technology you use is not as important as the company culture or processes to facilitate DevOps, but using that technology incorrectly will severely limit your efforts.

So while most of us understand what Continuous Delivery is, and what the benefits are, there is still a lot of work to do to get DevOps working for the entire enterprise. Let's get started.

BY MATT WERNER
CONTENT AND COMMUNITY MANAGER
RESEARCH@DZONE.COM

TABLE OF CONTENTS

PRODUCTION
CHRIS SMITH
DIRECTOR OF PRODUCTION

ANDRE POWELL
SR. PRODUCTION COORDINATOR

G. RYAN SPAIN
PRODUCTION PUBLICATIONS EDITOR

ASHLEY SLATE
DESIGN DIRECTOR

MARKETING
KELLET ATKINSON
DIRECTOR OF MARKETING

LAUREN CURATOLA
MARKETING SPECIALIST

KRISTEN PAGÀN
MARKETING SPECIALIST

NATALIE IANNELLO
MARKETING SPECIALIST

EDITORIAL
CAITLIN CANDELMO
DIRECTOR OF CONTENT + COMMUNITY

MATT WERNER
CONTENT + COMMUNITY MANAGER

MICHAEL THARRINGTON
CONTENT + COMMUNITY MANAGER

MIKE GATES
CONTENT COORDINATOR

SARAH DAVIS
CONTENT COORDINATOR

TOM SMITH
RESEARCH ANALYST

BUSINESS
RICK ROSS
CEO

MATT SCHMIDT
PRESIDENT & CTO

JESSE DAVIS
EVP & COO

MATT O'BRIAN
DIRECTOR OF BUSINESS DEVELOPMENT
sales@dzone.com

ALEX CRAFTS
DIRECTOR OF MAJOR ACCOUNTS

JIM HOWARD
SR ACCOUNT EXECUTIVE

JIM DYER
ACCOUNT EXECUTIVE

ANDREW BARKER
ACCOUNT EXECUTIVE

CHRIS BRUMFIELD
ACCOUNT MANAGER

ANA JONES
ACCOUNT MANAGER

WANT YOUR SOLUTION TO BE FEATURED IN COMING GUIDES?
Please contact research@dzone.com for submission information.

LIKE TO CONTRIBUTE CONTENT TO COMING GUIDES?
Please contact research@dzone.com for consideration.

INTERESTED IN BECOMING A DZONE RESEARCH PARTNER?
Please contact sales@dzone.com for information.

SPECIAL THANKS
to our topic experts, Zone Leaders, trusted DZone Most Valuable Bloggers, and dedicated users for all their help and feedback in making this guide a great success.

Executive Summary

BY MATT WERNER
CONTENT AND COMMUNITY MANAGER, DZONE

Ever since DevOps became a permanent topic of discussion in the world of software development, there's been a push from thought leaders, developers, and businesses to adopt the tools and methodologies to achieving Continuous Delivery. The benefits are obvious: with increased collaboration between development and operations teams, automated testing and deployments, and a supportive culture, bottlenecks can be eliminated, code changes can be deployed faster, and applications can recover from downtime much quicker. This year, it seems that understanding around the processes, and benfits is starting to make a substantial difference in organizations and their employees, compared to previous years. For *DZone's 2017 Guide to DevOps: Continuous Delivery and Automation*, we surveyed almost 500 DZone members about how their organizations have adopted DevOps, what their pain points are, what tools they use, and how they've finally made real headway to implementing Continuous Delivery throughout their organizations.

PROGRESS IS BEING MADE TO ACHIEVE CONTINUOUS DELIVERY

DATA 41% of survey respondents reported that their organization had a DevOps team, compared to 34% in 2016. Between 2017 and 2016, there was a 9% increase in those who felt they had adopted CD for some projects, while respondents who felt Continuous Delivery was a focus of their organization increased by 8%.

IMPLICATIONS Based on the growth in those who are doing continuous delivery for some projects, or have made it a focus for their team, it's clear that more developers and managers have been able to at least start their journey towards Continuous Delivery. There's also been an increase in education around what constitutes Continuous Delivery. These huge gains reflect organizations who recognize they haven't fully implemented Continuous Delivery yet, which means they understand what full Continuous Delivery means across the whole organization.

RECOMMENDATIONS For all the progress in partially adopting Continuous Delivery, there still seems to be a lack of knowledge around Mean Time to Recovery and Mean Time Between Failures, so developers need to communicate more with their ops teams to figure out how long their services are down, then come together and figure out a solution to the problem. While the increase in DevOps teams can be a step in the right direction, some experts feel that the existence of a "DevOps team" is not the same as practicing DevOps. In their quest to achieve Continuous Delivery, organizations and teams should continue to automate whatever processes they can.

TECHNOLOGY MATTERS FOR RECOVERY RATES

DATA Organizations that use push-button or automated deployments recover from failure twice as fast on average, compared to organizations that do not have such tools (12 hours vs 24). Container users found that, on average, it took 20% less time to recover than organizations that do not use containers. Those who have adopted microservices architectures for their applications have a 7-hour Mean Time to Recover (MTTR), compared to a 29-hour MTTR for those who don't.

IMPLICATIONS Continuous Delivery may not always be about tooling, but it is clear that tools help achieve the goals of CD. Modular, scalable technology, in addition to automated deployments, are key to reducing application downtime. In particular, the data suggests there is substance to the well-known hype around container technology, such as Docker, and microservices architectures. As mentioned in the "recommendations" in the previous sections, automation is incredibly important to quickly push changes to production and to get applications back online as soon as possible.

RECOMMENDATIONS If your organization or team can implement microservices or containers into a new or existing application, the data suggest that it's worth it to give them a try. MTTR is a very important metric to track and decrease, as large-scale, long-term failures will only produce angry users, and ultimately less revenue for your organization.

TEARING DOWN THE WALLS

DATA Compared to last year's list of barriers to adopting Continuous Delivery, survey respondents noted there were significantly fewer blockers. In particular, 7% fewer DZone members reported that there was no support from management, and 5% fewer respondents believed that engineers did not possess the skills to implement Continuous Delivery. All other barriers decreased 2-4% between last year and this year.

IMPLICATIONS There seems to have been a major shift in company culture, and while management is still identified as the greatest blocker to adopting DevOps, more managers have learned about the benefits of CD and have encouraged their teams to pursue it. This can also be reflected in the increase in DevOps adoption for some projects. In addition, with all the open source tools available, such as Jenkins, and wealth of knowledge on the topic available online, more developers are learning the right skills.

RECOMMENDATIONS Developers and managers who understand the benefits of DevOps need to educate both their managers and employees on the economic and engineering benefits of Continuous Delivery. Continuous Delivery needs a supportive culture to thrive in, and those supportive cultures come from a mix of enthusiasm from both management and employees, which helps to create an understanding of shared goals. See our infographic on page 16 for more information on how these barriers, and others, can block DevOps adoption.

Key Research Findings

BY G. RYAN SPAIN
PRODUCTION COORDINATOR, DZONE

497 respondents completed our 2017 Continuous Delivery survey. The demographics of the respondents include:

- 19% of respondents work at organizations with over 10,000 employees; 20% at organizations between 1,000 and 10,000 employees; and 26% at organizations between 100 and 1,000 employees.
- 45% of respondents work for organizations headquartered in Europe, and 30% for organizations based in the US.
- On average, respondents had 15 years of experience as IT professionals; 27% had 20 years or more of experience.
- 42% of respondents identified as developers/ engineers, and 27% identified as developer team leads.
- 82% of respondents work at companies using the Java ecosystem, and 70% at companies using client-side JavaScript.

TIME TO FACE THE STRANGE

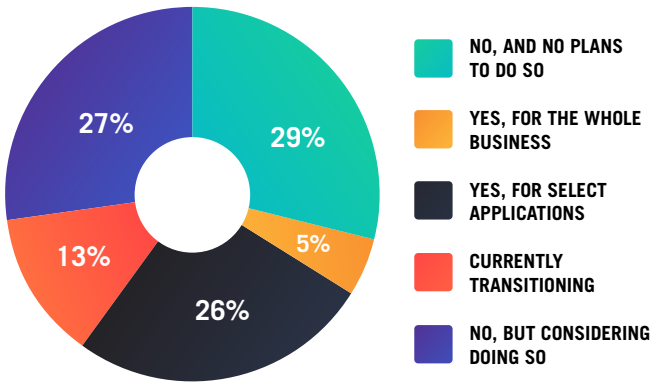
DevOps has had some steady growth over the past year, as more and more developers and organizations work towards automation and cross-departmental collaboration. 41% of respondents said their organization has a dedicated DevOps team, up 7% over last year's statistic (which had not changed from the year before). Performance issue detection in the software delivery process increased 5% year over year, while automated performance testing increased 6% and automated feature validation increased 4%. The number of respondents who said they believe their organization has achieved Continuous Delivery "for some projects" increased 9% from 2016, and there was an 8% swing in respondents who said that CD is a focus for their organization. Microservice architectures are used 7% more compared to last year, and container adoption is up 8%. The use of version control tools reported in QA and Production have increased 15% and 18% respectively, and the use of CI tools in those departments increased 17% and 13%.

Despite this growth, there are some areas of stagnation in CD results. From 2016, there was no statistically significant change in respondents' estimate of their mean time to recovery (between hours and days) or mean time between failures (between hours, days, and months). Most CD pipeline pain points also remained the same from last year, with the exception of automated testing, which dropped 7% as a pain point, and the deployment process and regression testing, which each dropped 4%.

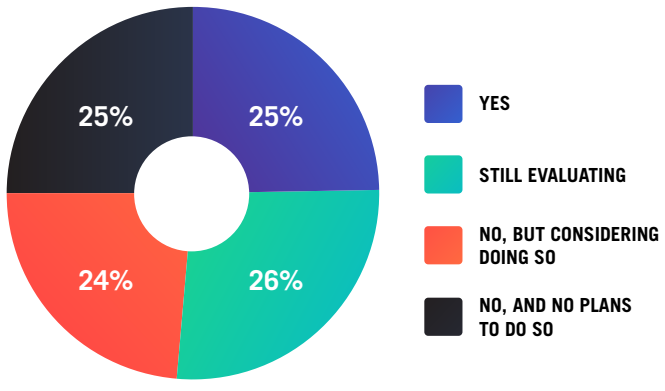
SIZE MATTERS

With regards to having Continuous Delivery implemented in an organization, and having Continuous Delivery implementation be a focus for an organization, company size plays a sizable role. Respondents' belief that their company has achieved Continuous Delivery trends upwards as the size of their organizations increase. 51% of respondents working at companies under 100 employees think their company has either fully or partially achieved Continuous Delivery, versus 60% of respondents who work at companies larger

HAS YOUR ORGANIZATION MOVED TOWARDS A MICROSERVICE ARCHITECTURE?



HAVE YOU OR YOUR ORGANIZATION ADOPTED CONTAINER TECHNOLOGY (E.G. DOCKER)



than 10,000 employees. And only 41% of respondents working at sub-100 employee organizations say that CD is a focus for their company, 15% less than those who work at companies between 100 and 9,999 employees and 30% less than those who work at 10,000+ organizations.

This goes hand-in-hand with larger companies' ability, and likely need, to have dedicated DevOps teams. Only about one in four respondents (27%) at an organization with fewer than 100 employees said their company had a dedicated DevOps team, compared to almost half (45%) of respondents between 100 and 9,999 employees and 62% over 10,000.

STOP... RECOVERY TIME

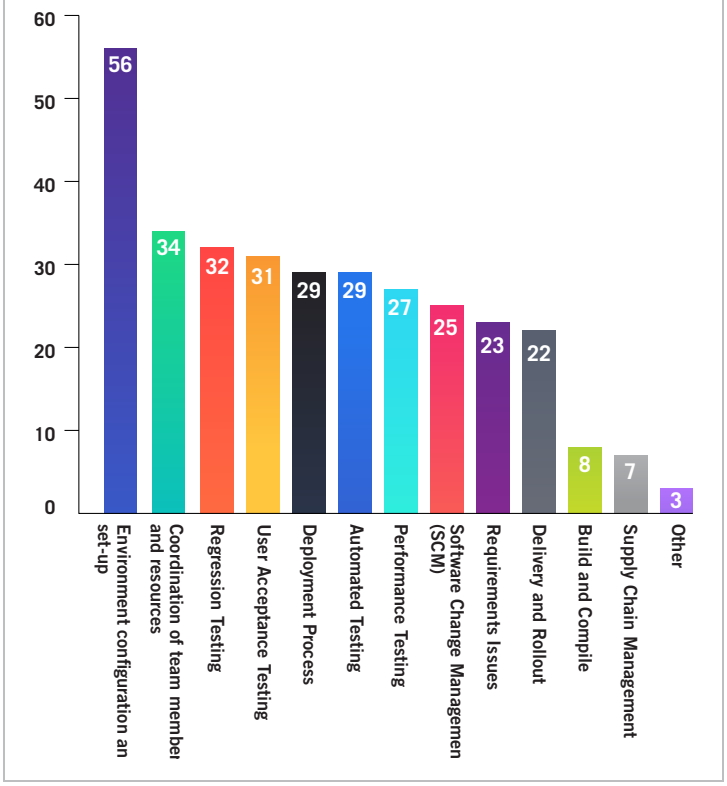
Overall, respondents' estimated an average mean time to recovery of just under 19 hours, with estimates ranging from just minutes to 40 days, but with most estimates falling somewhere between 2 and 24 hours. Several factors come into play here, which can drastically change the mean-time-to-recovery estimates. Respondents whose organizations have push-button/automated deployment estimated recoveries happen twice as fast as organizations that don't (12 hours versus 24 hours). Those respondents using or evaluating container technologies estimated about 20% less time to recover than non-container users (17 hours versus 21 hours). Microservice usage greatly affected these estimates. Respondents who said their organization has not moved to microservice architectures estimated, on average, a 29-hour mean time to recovery; respondents at organizations currently transitioning estimated 12 hours; and respondents at organizations using microservices for some or all of their applications estimated a 7 hour mean time to recovery.

BRING THE PAIN POINTS

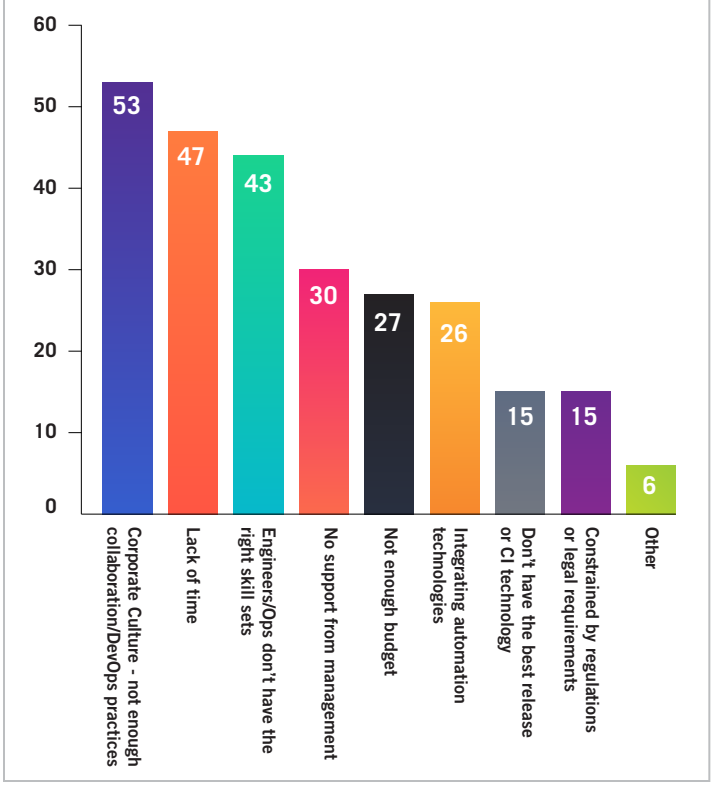
We asked our survey-takers who said they believed their organization had implemented Continuous Delivery in some capacity what their biggest pain points were in the CD pipeline, and likewise asked those respondents who did not think their organization had achieved CD status what they thought were the main barriers to adopting CD. As mentioned earlier, most pain points appeared to be just as painful this year as they were last year. The most common pain points were environment config and setup (56%), coordination of team members and resources (34%), and regression testing (32%). Most other pain points were experienced by roughly a quarter of respondents, with the exceptions of build and compile (8%) and supply chain management (7%).

Regarding barriers to adoption, this year's respondents again answered similarly to last year's results, though all barriers did drop somewhat. The biggest changes here were "no support from management," which dropped 7% from last year, and "Engineers/Ops don't have the right skill sets," which dropped 5%. All others dropped between 2 and 4 percent from last year. So, while progress is being made to make CD easier to adopt and manage, there is still certainly plenty of room for improvement.

WHAT ARE YOUR BIGGEST PAIN POINTS IN THE CONTINUOUS DELIVERY PIPELINE?



WHAT ARE YOUR MAIN BARRIERS TO ADOPTING CONTINUOUS DELIVERY?



Continuous Delivery Anti-Patterns

BY **MANUEL PAIS AND MATTHEW SKELTON**
SKELTON THATCHER CONSULTING

QUICK VIEW

01 Single vendor "DevOps one-stop shop" solutions trade ownership of your toolchain for ease of setup and slow you down over time.

02 Cost of lack of evolvability is often underplayed, while cost of individual tool integration is overrated.

03 Out-of-the-box tools also fail. A combination of poor error messages and lack of access to logs leads to massive waste of time.

04 Tools that equal a stage to an environment deployment miss out on the real power of deployment pipelines.

05 Out-of-the-box solutions propagate vendors' misunderstandings of principles and practices.

An anti-pattern is typically a behavior that hinders or at least negatively influences a target objective. There are many anti-patterns for Continuous Delivery documented by people like Jez Humble, Steve Smith, and ourselves.

Some CD anti-patterns arise not from an existing behavior, but by trying to actually implement the practices without understanding their goals first. That is especially frequent when adopting integrated tools that are supposed to support a full set of CD practices out-of-the-box. The idea that you can "just plug in a deployment pipeline" and get the benefits of CD is a common anti-pattern.

In the following sections we will look into some of these "unintentional" and often hard-to-notice anti-patterns that might emerge when adopting a "DevOps one-stop shop" solution.

ANTI-PATTERN #1: AVOIDING TOOL INTEGRATION LIKE THE PLAGUE

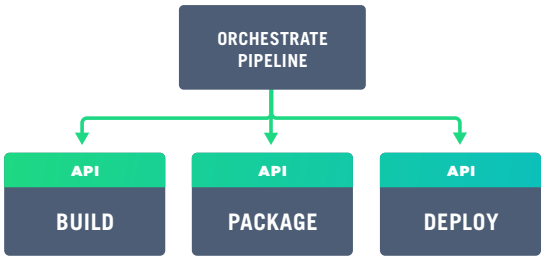
There is a prevalent notion, especially in larger enterprises, that integrating disparate tools is extremely expensive. That you'll be locked for eternity maintaining glue code with high technical debt. That might have been true in the 2000s, but surely not today.

As long as you are integrating tools with clear and standard APIs, the orchestration code can be minimal and understandable by anyone familiar with API development (which all developers should be in 2017!).

Tooling integration cost is not zero, but it's negligible when compared to the potential cost of **not** integrating. One-stop solutions might embed erroneous concepts. Any tool might. The problem is that the former will propagate them across the entire lifecycle.

Anti-pattern: *Single tool without standard APIs, or only for some components.*

Instead, single-purpose, focused tools with a well-defined API help reduce the blast radius of bad practices. And you can swap them easily when they don't match your requirements anymore. A flexible toolchain standardizes practices, not tools. It supports certain capabilities, which are easy to locate and expand on, replacing particular components (tools) when required.



Pattern: *Single-purpose tools with clear APIs/boundaries that can be replaced more easily.*

Another gain with individual tools: you can actually expect an answer from the vendor when you ask for a feature, since they have a reduced feature set and faster change cycles. A vendor of a one-stop solution has a lot more requests in its backlog. Chances are, if you're not a major client, your requests will get buried for months or even years.

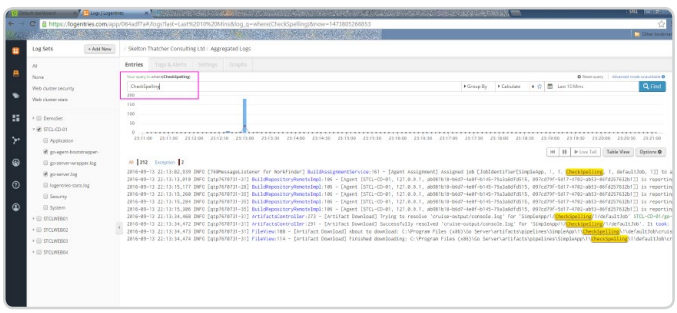
> "The idea that you can 'just plug in a deployment pipeline' and get the benefits of CD is a common anti-pattern."

ANTI-PATTERN #2: ERROR HANDLING & LOGGING BEHIND CLOSED DOORS

Another hidden time-consuming anti-pattern in one-stop solutions derives from generic error messages or inaccessible logging. This tends to be especially painful with SaaS solutions.

Unfortunately our industry is still plagued with the "abstraction everywhere bias": a tendency to favor generic error messages ("VM could not be started" or "Deployment failed") instead of spelling out what was the expected input or output and the difference to the actual result. Now add to the mix inaccessible logs for those failed operations, a common situation in one-stop solutions that provide only UI access to certain features or only let you access logs via queries.

The problem is these tools assume they have all the use cases and all the failure scenarios covered. That is untrue for any tool, or any software in fact. We will always need access to information to troubleshoot issues. The more information we have, the more likely we can correlate events and find the causes.



Pattern: *Centralizing logs from individual tools and correlating messages.*

Think of all the time spent deciphering error messages, trying to guess what went wrong, or waiting for a vendor's support to get back to you (if you hit the jackpot with a technical issue deep in the tool's gut, good luck waiting for the support-to-engineering return trip time). That time alone is an order of magnitude higher than any individual tool integration time you'd have spent.

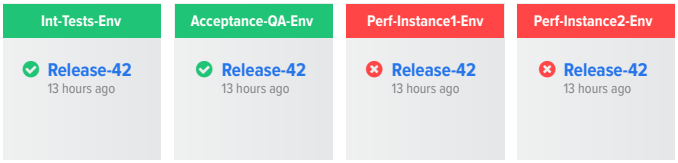
ANTI-PATTERN #3: ENVIRONMENT-DRIVEN PIPELINES

This one is pretty self-explanatory. Tools that assume your pipeline is nothing but a sequence of environments where you deploy your system and run some tests.

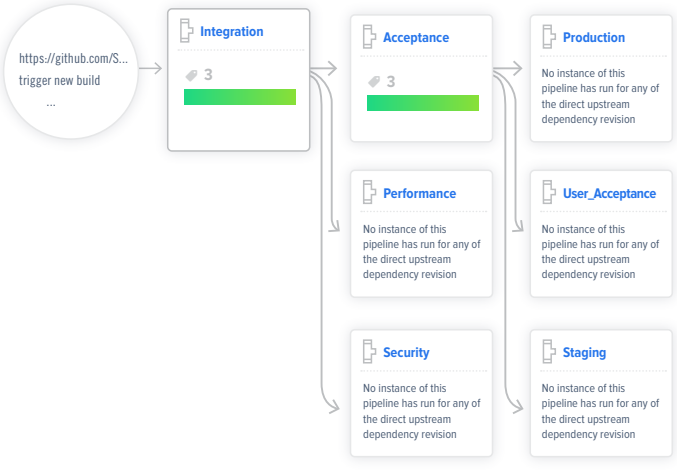
A pipeline stage represents an activity in the delivery chain. It might require:

- One environment: typical for acceptance tests
- Zero environments: typical for manual checks/analysis or approval requests
- Multiple environments: typical for performance tests

Thus pipeline stages should not be tightly coupled to environments. Assuming only the first option above leads to pipelines that contemplate only automatable activities, hiding other (often non-technical) activities that are part of delivery as well. In turn, this leads to lack of visibility on (real) bottlenecks and local optimization (technical steps) instead of global (cycle time).



Anti-pattern: *Pipeline stages = Environment deploys*



Pattern: *Pipeline stages represent activities, which might require 0 to many environments.*

ANTI-PATTERN #4 FLEXING IS FOR FITNESS, NOT FOR PRINCIPLES

Adopting core Continuous Delivery principles is hard and often requires mental and cultural shifts. Without them the

underlying practices become ceremonies, instead of actual improvements in delivery. Flexibility is fine for the gym, but not for core principles required for a (often radical) new way of working.

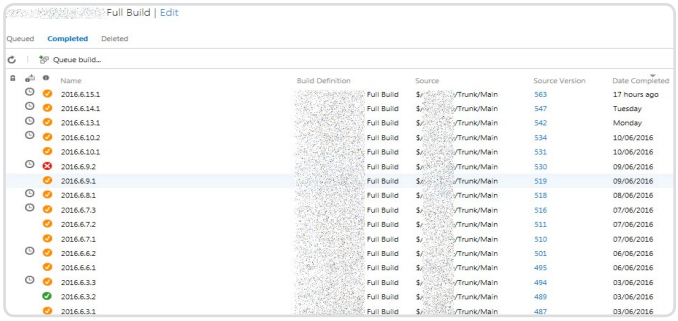
If the tools supporting the practices do not align with the principles, they end up unintentionally undermining the whole endeavor. In contrast, working with a set of single-purpose tools helps identify and address erroneous assumptions by any one tool.

Below are a couple of examples of misalignment between implementation and principles that we've seen in some out-of-the-box integrated tools.

STATUS: ???

A pipeline status should be binary. Red or green. Not orange. Not grey. Not blue. Recurring ambiguities in status inevitably lead to disengagement by development teams. This is the CD equivalent to warnings at compilation time. If the first warnings are ignored by developer A, then developer B and developer C will ignore them as well. Soon everyone just assumes having 372 warnings is ok.

Having an uncontested pipeline status is a prerequisite to the Continuous Delivery principle of "stopping the line" when a pipeline fails (then either fix it quickly or revert the changes that broke it). Interestingly, this is also a prerequisite to get rid of those nasty compilation warnings (try making the pipeline go red if there are compilation warnings).

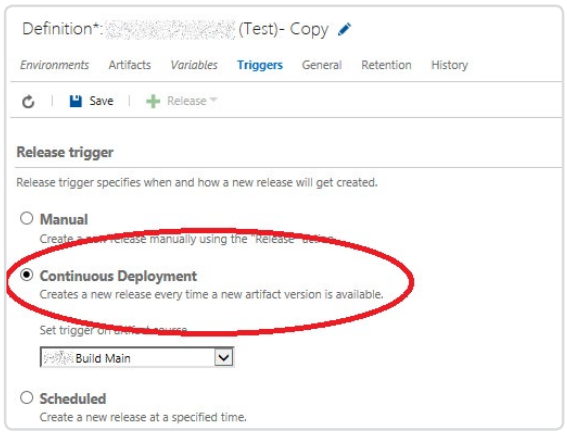


Anti-pattern: *Non-binary status: once upon a time there was a green build...*

TERMINOLOGY FAIL

Another plague in our industry is the proliferation of terminology. We have enough confusion as it is and quite frankly one-stop tool vendors are not helping. They, above all, should strive to align on common terminology, as they are informing their clients on the entire lifecycle. So it better be correct. This is clearly complicated as those vendors have many different teams working on the integrated tools. But it is needed.

One puzzling example we have come across of terminology failures is calling a pipeline trigger from a successful build a "continuous deployment."



Anti-pattern: *Using common terminology to represent something totally unrelated.*

Another example are "release definitions" instead of "pipeline definitions" (the image above is a release definition configuration). Legacy terminology leads to legacy behaviors, thinking of releases and work batches instead of pipelines and frequent delivery of small, low-risk changes in production.

This might seem like just nitpicking, but the accumulation of all these misunderstandings leads to unknowingly misinformed organizations and teams.

SUMMARY

We want to go faster and faster delivering products. To do so we should also be able to go faster and faster in adapting our pipeline to support that goal.

But one-stop tooling solutions often bring along several anti-patterns that slow us down or misguide us, as explored in this article.

We're not advocating for always integrating your own tool chains. Your organization might have good reasons to go for a one-stop solution in terms of DevOps and Continuous Delivery. We just recommend being extremely conscious of the trade-off you're making. And we hope the anti-patterns highlighted in this article help guide some of that thinking process.

MANUEL PAIS is a people-friendly technologist at @SkeltonThatcher. He's a Continuous Delivery and DevOps advocate, with a diverse background as developer, build manager and QA lead. Also @InfoQ DevOps lead editor, co-organizer of @DevOpsLisbon meetup.

MATTHEW SKELTON has been building, deploying, and operating commercial software systems since 1998. Co- founder and Principal Consultant at Skelton Thatcher Consulting, he specializes in helping organizations to adopt and sustain good practices for building and operating software systems: Continuous Delivery, DevOps, aspects of ITIL, and software operability.

Key Functionalities of CI Tools

BY MANUEL MOREJON

DevOps teams should strive to create and support workflows that best fit the entire team. During workflow creation, it is important to evaluate your team's needs and think ahead when selecting a continuous integration tool. Sometimes, selecting a local CI solution is a necessary or a political requirement in teams or enterprises, so the installation and configuration of the system must be set up within the organization's own infrastructure. There are many CI systems that offer the ability to tie directly to local infrastructure, but not all do — for this checklist, we'll focus on the ones that do.

1. PIPELINE

Continuous Integration systems should have an interface showing the current process state and all task transitions during the deployment process. These systems should manage tasks in both the sequential and parallel. They also should give the option for the DevOps team to handle manual actions in order to specify when they need something to be deployed.

2. BUILD AS CODE

Whilst having every single mobile device on your desk would be pretty awesome, it's not really feasible in the real world. How about resizing your viewport instead? Chrome provides you with everything you need. Jump into your inspector and click the 'toggle device mode' button. Watch your media queries come to life!

3. WORK WITH CONTAINERS

Containers have proven to be of great importance in system development. Docker is a primary tool in the development process. Continuous Integration systems have been adding functionalities to manage containers and images as part of daily routines.

4. MULTI-PLATFORM

You should be able to execute constructions in Unix, Windows, and Mac OS X, allowing your team to develop from any operating system.

5. ACTIVE SUPPORT

Concepts like continuous integration and continuous deployment should be taken into account during the infrastructure tools selection. The systems selected should have short deployment cycles in order to guarantee quick bug fixes and new features from the team.

6. INTEGRATION WITH THIRD-PARTY RESOURCES

Allowing the integration between the system and external services is vital to the organization. The system itself doesn't need to have all functionalities, but it must have an architecture to allow increased benefit through third-party resources. Allowing support from the community has been demonstrated to be crucial in the improvement of the system.

Desirable Traits of Local CI Tools

Identifying the best tools for your organization must start from examining the needs of your development group. Below are some desirable traits to keep in mind when making your selection:

1. FREE UNLIMITED USERS

Allowing multiple users access to the systems and the use of role-based management for users.

2. FREE UNLIMITED PROJECTS AND BUILDS

Allowing management of multiple projects and unlimited execution of builds.

3. FREE UNLIMITED AGENTS

Allowing build executions on multiple machines.

Local CI Tools Analyzed

Each one of the tools listed here includes the features mentioned above.

1. GITLAB CE

GitLab CI is a part of GitLab that allows you to manage jobs with tasks through the ***.gitlab-ci.yml*** file. This file is used to describe all tasks in a single file with the YAML format.

GitLab CI allows for the use of Docker to build and deploy projects, and they offer a Private Docker Registry to store your own images. GitLab Runner can be installed cross-platform and is responsible for running jobs on external machines. GitLab CE updates its products the 22nd of every month.

2. JENKINS

Jenkins uses the Pipeline Plugin to manage and describe the deployment process. This plugin uses a ***Jenkinsfile*** to register all steps that should be executed by the Pipeline Plugin. Jenkins also has the Blue Ocean Plugin to improve user experience during the use of the Pipeline Plugin.

The Docker Plugin allows Jenkins users to manage topics related to containers and images. Jenkins uses node philosophy to run jobs on external machines. These machines can have different operating systems, allowing them to run cross-platform functionalities. Jenkins updates its products each month.

3. CONCOURSE CI

Concourse is a Continuous Integration tool created with the pipeline as a first-class citizen. The system shows us a new way to manage jobs in the server. Concourse uses files with the YAML format to describe jobs and tasks.

In Concourse, every process runs inside a container. Concourse workers can be installed cross-platform and they are responsible for running the jobs described in the YAML format inside containers. Concourse CI update its products each month.

Advantages of Using Local CI Tools

The advantages obtained when using a local CI system include:

1. THE CODE INTEGRATION IS CONSTANT

The development group is not as likely to be affected by any unplanned interruptions during integration of the code as they might if using an outsourced cloud-hosted system.

2. SYSTEM UPDATES ARE CONTROLLED

Updates sometimes involve making adjustments to project settings. An unwanted update may cause errors in code integration—stay in control of updates by using a local tool.

3. GREATER PRIVACY OF WORK GROUP PRODUCTS

Although there are security and privacy policies when using paid cloud services, it is always best to leave the code at home behind a firewall.

Disadvantages of Using Local CI Tools

The tools analyzed in this article can be acquired without any cost, but this does not mean that the installation and the maintenance are totally free for the development team. The team must invest in:

1. HARDWARE INFRASTRUCTURE

Physical or virtual machines with medium or high profile.

2. SPECIALISTS IN THE AREA

Engineers and Specialists with the correct knowledge.

MANUEL MOREJON is a Master in Applied Software, a DevOps Engineer, and a Configuration Manager. Manuel has both the technical and communication skills to help teams improve their workflows and reduce deployment times, the goal always being to maximize productivity by minimizing errors. Enthusiastic about teaching and sharing knowledge with the community, Manuel writes at mmorejon.github.io/en/blog.

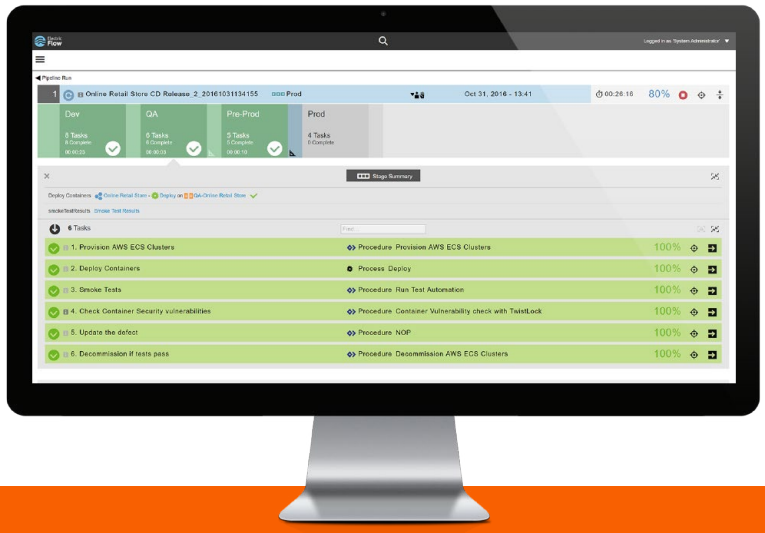
Electric Cloud

Accelerate Dev. Simplify Ops. Deliver Faster.

FREE!

ElectricFlow: *The Most Advanced DevOps Release Automation Solution*

- **Plug-in all point-tools, stacks and processes** to orchestrate pipelines from check-in to Production
- **Deploy to any cloud or container environment** without heavy scripting or learning new APIs
- **Powerful DSL, CLI and APIs** provide programmatic access to all UI functionality



USE IT FREE: electric-cloud.com/electricflow

The Most Advanced DevOps Release Automation Solution

For over a decade, Electric Cloud has been helping organizations automate and accelerate their software delivery. ElectricFlow, the company's end-to-end DevOps Release Automation platform, has recently been named the leader in Gartner's Magic Quadrant for Application Release Automation—getting the highest scores for both vision and execution.

ElectricFlow orchestrates and automates the entire software delivery pipeline. By allowing customers to plug in all the point-tools, environments, and processes involved in their delivery process, ElectricFlow enables organizations to orchestrate the end-to-end pipeline and achieve visibility, compliance, and control over the entire release process.

With ElectricFlow, enterprises can model their application delivery pipelines, to normalize their "Pathways to Production" as much as possible to allow faster feedback loops and reusability across the organization, while supporting the specific needs of different teams or variants of the application. The ability to have consistent build, test, and deployment processes that everyone benefits from is the "Holy Grail" for enterprises as they take the next steps in their DevOps adoption towards scaling DevOps in the organization.

By standardizing on ElectricFlow to manage their tools and processes across the organization, customers eliminate configuration drift, bottlenecks in the pipeline, and save on operational cost and management overhead. The tight integration between the various processes and tools eliminates manual handoffs or silos of automation to speed up cycle times, provide better product quality, and improve resource utilization.

WRITTEN BY ANDERS WALLGREN
CTO, ELECTRIC CLOUD, INC.

PARTNER SPOTLIGHT

ElectricFlow By Electric Cloud

Electric Cloud

Accelerate Dev. Simplify Ops. Deliver Faster.

CATEGORY
DevOps Release Automation

NEW RELEASES
Every 60 days

OPEN SOURCE
No, but we offer a free community edition

STRENGTHS

- Automating and accelerating software delivery since 2002
- Broadly recognized by leading analyst as DevOps Release Automation Leader
- Single, scalable DevOps Release Automation platform that shrinks cycle times
- Automate and orchestrate your entire end-to-end pipeline—plug in all your tools, stacks & processes
- Deploy to any cloud/container environment without heavy scripting or learning new APIs

CASE STUDY

Companies like E-Trade, HPE, Gap, and Huawei trust Electric Cloud with their DevOps Release Automation for faster releases, fewer process errors, better visibility, and better software quality. For Huawei, ElectricFlow is part of a comprehensive solution, supporting more than 2000 releases per year, 50K compile & builds per day, 100 million test cases run per day including more than 30 million LOC for a very complex product. They completed more than 480K code reviews/analyses per year and more than 170K system integration tests per year. E-Trade uses ElectricFlow for faster build, test, and deploy cycles for their website, mobile clients, trading engine, and settling systems. Since using ElectricFlow, they've experienced 12x faster delivery.

NOTABLE CUSTOMERS

- E-TRADE
- GE
- Intel
- Gap
- HPE
- Lockheed Martin

BLOG electric-cloud.com/blog

TWITTER @electriccloud

WEBSITE electric-cloud.com

9 Benefits of Continuous Integration:

Why Modern Development Teams Should Automate Testing

BY **JOB VAN DER VOORT**
VP OF PRODUCT AT **GITLAB**

QUICK VIEW

01 Continuous Integration, Continuous Deployment, and Continuous Delivery together enable a team to safely build, test, and deploy code.

02 Automating testing through Continuous Integration is a way to increase code quality.

03 9 benefits of Continuous Integration are listed, explained, and help shed light on why teams should adopt these practices.

Across almost every industry, the best companies are increasingly becoming those who make great software. As a result, software development teams are transforming their software operations to optimize for efficiency, quality, and reliability. Continuous Integration, Continuous Deployment, and Continuous Delivery are increasingly popular topics among modern development teams. Together they enable a team to safely build, test, and deploy their code. These modern development practices help teams release quality code faster and continuously.

CONTINUOUS INTEGRATION, DELIVERY, AND DEPLOYMENT DEFINED

Continuous Integration is a software development practice in which you **build and test** software every time a developer pushes code to the application.

For example, let's say developers push code to improve their software projects every day, multiple times per day. For every commit, they use a CI tool to test and build their software. The CI tool will run unit tests to make sure their changes didn't break any other parts of the software. Every push automatically triggers multiple tests. Then if one fails it's much easier to identify where the error is and start working to fix it. But for this team, they do not deploy to production, so this is considered Continuous Integration only.

Continuous Delivery is a software engineering approach in which **continuous integration, automated testing,** and **automated deployment** capabilities allow software to be developed and deployed rapidly, reliably, and repeatedly with minimal human intervention. Still, the **deployment to production** is defined strategically and triggered manually.

Mozilla is a good example of an organization using Continuous Delivery. Mozilla says that for many of their web projects "once a code change lands on a master branch it is shepherded to production with little-to-no human intervention."

Continuous Deployment is a software development practice in which every code change goes through the entire pipeline and is put **into production automatically**, resulting in many production deployments every day. It does everything that Continuous Delivery does, but the process is fully automated; there's **no human intervention at all**.

Hubspot, Etsy, and Wealthfront all use continuous deployment to deploy multiple times a day. In 2013, Hubspot reported that they deploy 200-300 times day. People often assume that continuous deployment only works for web-based software companies, so I'd like to offer another example in a completely different industry: Tesla. Tesla Model S is using continuous deployment to ship updates to the firmware on a regular basis. These changes don't simply change the dashboard UI or offer

new ways to change console settings in your car, they improve key elements of the car, like acceleration and suspension. Tesla proves that continuous delivery can work for any team committed to the practice.

9 BENEFITS OF CONTINUOUS INTEGRATION

The benefits of Continuous Integration, Delivery, and Deployment are clear. However, the process to building the pipeline to do Continuous Delivery or Deployment isn't always easy. But that doesn't mean you shouldn't do it. We believe that modern development teams are working their way up to Continuous Deployment. That's why we suggest getting started with automated testing through Continuous Integration today. With automated tests on each new commit, Continuous Integration is a great way to increase code quality. Here are our top nine reasons why we think every development team should be doing Continuous Integration.

1. MANUAL TESTS ARE ONLY A SNAPSHOT

How many times have you heard a team member say "it worked locally." In their defense, it likely did work locally. However, when they tested it locally they were testing on a snapshot of your code base and by the time they pushed, things changed. Continuous Integration tests your code against the current state of your code base and always in the same (production-like) environment, so you can spot any integration challenges right away.

2. INCREASE YOUR CODE COVERAGE

Think your tests cover most of your code? Well think again. A CI server can check your code for test coverage. Now, every time you commit something new without any tests, you will feel the shame that comes with having your coverage percentage go down because of your changes. Seeing code coverage increase over time is a motivator for the team to write tests.

3. INCREASE VISIBILITY ACROSS THE TEAM

Continuous Integration inspires transparency and accountability across your team. The results of your tests should be displayed on your build pipeline. If a build passes, that increases the confidence of the team. If it fails, you can easily ask team members to help you determine what may have gone wrong. Just like code review, testing should be a transparent process amongst team members.

4. DEPLOY YOUR CODE TO PRODUCTION

A CI system can automatically deploy your code to staging or even production if all the tests within a specific branch are green. This is what is formally known as Continuous Deployment. Changes before being merged can be made visible in a dynamic staging environment, and once they are merged these can be deployed directly to a central staging, pre-production, or production environment.

5. BUILD STUFF NOW

All your tests are green and the coverage is good, but you don't handle code that needs to be deployed? No worries! CI servers can also trigger build and compilation processes that will take care of your needs in no time. No more having to sit in front of your terminal waiting for the build to finish, only to have it fail at the last second. You can run any long-running processes as a part of your CI builds and the CI system will notify you if anything goes wrong, even restarting or triggering certain processes if needed.

6. BUILD STUFF FASTER

With parallel build support, you can split your tests and build processes over different machines (VMs/containers), so the total build time will be much shorter than if you ran it locally. This also means you'll consume fewer local resources, so you can continue working on something else while the builds run.

7. NEVER SHIP BROKEN CODE

Using continuous integration means that all code is tested and only merged when all tests pass. Therefore, it's much less likely that your master builds are broken and broken code is shipped to production. In the unlikely event that your master build is broken, let your CI system trigger a warning to all developers: some companies install a little warning light in the office that lights up if this happens!

8. DECREASE CODE REVIEW TIME

You can have your CI and Version Control System communicate with each other and tell you when a merge request is good to merge: the tests have passed and it meets all requirements. In addition, even the difference in code coverage can be reported right in the merge request. This can dramatically reduce the time it takes to review a merge request.

9. BUILD REPEATABLE PROCESSES

Today's pace of innovation requires development teams to deliver high-quality software faster than their competition. Modern development teams are building efficient software delivery engines by creating repeatable processes that standardize development best practices. With automated testing, your code is tested in the same way for every change, so you can trust that every change is tested before it goes to production.

JOB VAN DER VOORT discovered his love for building software while working in neuroscience and quickly left academia for GitLab. As the VP of Product at GitLab, Job is responsible for building software that helps *anyone* go faster from idea to production. In his free time, Job likes to build apps, play board and video games, and explore new tech.

Let's Automate Business.

Automic™

Automic, the leader in Business Automation, helps enterprises drive competitive advantage by automating their IT and business systems - from on-premise to the Cloud, Big Data and the Internet of Things. With offices worldwide, Automic powers over 2,700 customers including Bosch, Netflix, eBay, AMC Theatres, Carphone Warehouse, ExxonMobil, Vodafone, Société Générale, NHS SBS, General Electric and Swisscom. More information can be found at www.automic.com.

Automic™.com
@automic

Automic is the largest global pure-play automation solution provider. With a focus on customer satisfaction, continuous innovation, dedicated specialized teams and partnerships, we lead the business automation scene. We help over 2,700 companies - from start-ups to global brands - grow their business by taking away the stress of the every day so they can focus on innovation.

Automic has mastered the art of connecting islands of automation. We have proven this over and over again - significantly reducing complexity by orchestrating business processes with the broadest range of integrations.

We believe in building long lasting relationships with our customers and going the extra mile. To be our client's trusted partner requires interacting with clients on their most critical automation challenges. It's not just our differentiated product features that make the difference, but also our dedicated automation specialists located in each region, as close as possible to our clients. We focus on Business Automation.

> We help companies grow their business by taking away the stress of the every day so they can focus on innovation.

We have over 20 years of experience in this field. That's how we do business - we empower business through automation.

WRITTEN BY LUCAS CARLSON
SENIOR VICE PRESIDENT (SVP) OF STRATEGY, **AUTOMIC SOFTWARE**

PARTNER SPOTLIGHT

Automic v12 By Automic

Automic

Automic V12 is a unified suite of business automation products which drive agility across Enterprise Operations and empower your DevOps initiatives.

CATEGORY
Release Automation / Workload Automation / Service Orchestration / DevOps, AgileOps

NEW RELEASES
Bi-annually

OPEN SOURCE
Yes

CASE STUDY

Working closely with Automic, an international wholesaler and retailer of jewelry transformed a department of project managers, developers and operations, responsible for all of the customer-facing websites, into a DevOps organization. As a result, project delivery was completed with 90 percent fewer delays, and first customer-facing releases of projects could be delivered in less than half the time. The usability of its websites increased significantly; resulting in more customer traffic, longer average time spent on the sites and increased revenue.

NOTABLE CUSTOMERS

- Bosch
- Netflix
- AMC Theatres
- ExxonMobil
- Vodafone
- NHS SBS
- General Electric

STREAMLINING DEV AND OPS PRODUCTIVITY BY:

- Switching the development from a project to a product mode (cross-project development teams based on capabilities rather than assignment of individual developers to projects).
- Moving project managers from development micromanagement to focus on major milestones and deliverables to the business.
- Enabling some project managers as product owners (project management remained necessary).
- Providing developers with agile methodologies through initial training plus ongoing coaching (six to 12 months).
- Providing ops with an agile methodology.
- Organizing both dev and ops into multi-skilled teams focused on individual products rather than technologies.
- Aligning ops into delivery teams, initially part-time and later full-time.
- Giving delivery teams the end-to-end responsibility for their products, including their operations.
- Standardizing the technology stacks within teams (but not across them) over time.
- Delivering projects into production (or alpha, beta) in slices (releases) instead of just once to get early feedback and improve.

BLOG automic.com/blog | **TWITTER** @automic | **WEBSITE** automic.com

HOW TO AVOID DEAD-END DELIVERY

In this year's survey of DZone's audience, 48% of respondents believe they have not adopted Continuous Delivery, and 38% believe they have adopted Continuous Delivery only for some projects. Just over half of respondents (54%) are currently focused on implementing Continuous Delivery in their companies, so what's keeping them from reaching that goal, and what's keeping the other 46% from trying to implement it? Turns out, there are a lot of obstacles that can prevent developers or managers from making headway in their adoption efforts. To learn more about them, we're going to play a little game...

Imagine you're a plucky young startup with everything to prove, or perhaps part of a seasoned corporation that's been around the block and is ready for a transition to more modern methodologies. Can you achieve Continuous Delivery without running into any of these barriers? A-maze us!

NO SUPPORT FROM MANAGEMENT

While the benefits of Continuous Delivery are well-documented, the initial investment into tooling and training can put a lot of managers off the concept. For successful Continuous Delivery, it takes both management and frontline developers to believe in the benefits and be devoted to working towards them.

LACK OF SKILL

Continuous Delivery is very difficult without adopting several new tools, and impossible without changing processes. Learning all these new technologies can be incredibly difficult, especially if there's no prior knowledge on your team.

CORPORATE CULTURE

Company culture can be difficult to establish, and even more difficult to change. If a culture has built silos that separate teams from each other, it's going to be very difficult to foster the collaboration, flexibility, and speed that Continuous Delivery demands.

LACK OF TIME

Jamie Zawinski once famously said, "Linux is only free if your time has no value." Unfortunately, in the Enterprise, whether you go for an open source or proprietary tool, implementing DevOps tools and processes take a lot of time that you may not have, especially if you have delivery dates looming.

INTEGRATING AUTOMATION TECHNOLOGY

The knowledge to put the pieces of your build pipeline together may not exist in your organization, and even if it does it could take a lot of work to integrate these tools, especially if those tools are open source and you don't have budget to spring for a proprietary product.

LACK OF BUDGET

If your organization doesn't have time to go the open source route, you'll need to use proprietary solutions, which you may not have the budget for, especially if you're a startup without VC or time to spare. No money, mo' problems.

Better Code for Better Requirements

BY **ALEX MARTINS**

CTO/ADVISOR - CONTINUOUS QUALITY AT **CA TECHNOLOGIES**

QUICK VIEW

01 Code may be of the highest quality, but if it's not reflecting what was specified in the requirements, you may have built perfectly useless code.

02 By preventing defects from being written into the code, quality is thus built into the application from the onset.

03 The use of a CAD-like tool in software engineering not only accelerates the software lifecycle, but also ensures developers are building the right things.

Quality is a very hot topic in the DevOps and Continuous Delivery era. "Quality with speed" is the theme of the hour. But most development and testing teams have different views on what quality means to them.

Looking back at my days as a professional developer, I remember being tasked to follow the company coding style guide. This described the design principles and the code convention all the developers should follow so that we wrote consistent code. Thus when a change request came in, anyone could read the code and make the edits, and we could minimize maintenance.

Then there were the weekly reviews where we would get together with a peer and go through the code to ensure we understood it and were following the style guide. If the code checked out, we then thought we had quality code.

But did that mean the applications we built were high quality? No!

SETUP

I've worked with plenty of agile dev teams that have adopted DevOps and achieved Continuous Delivery. These teams typically create basic, sometimes throwaway code just so they can quickly push a build out to users to get feedback and make quick adjustments. Of course this approach generates technical debt; however, at this stage, speed is more valued than code that is perfectly written according to any style guide.

Upon seeing positive feedback from users, these teams start constantly refactoring the code to keep technical debt at manageable levels. Otherwise, all the speed they've gained to roll out the first builds is lost as the code grows and becomes hard to change due to the technical debt accrued. The ultimate consequence: team capacity and velocity for future iterations is decreased, taking everyone back to square one – with not only less-than-adequate code, but also an application that users don't like.

So to keep improving their code in such a mature environment, these teams use code quality tools to profile the code and determine where to focus refactoring efforts first. This helps them **build things right**. But no matter how good the code gets, the user may still think the application sucks, simply because they were not **building the right things** in the eyes of the user. There is a difference between the two, and in my experience, this is a huge gap in most Continuous Delivery initiatives.

So what's the missing link? Requirements. The code may be of the highest quality, but if it's not reflecting what was specified in the requirements, you may have built perfectly useless code.

Louis Srygley has an apt description for this:

"Without requirements and design, programming is the art of adding bugs to an empty text file."

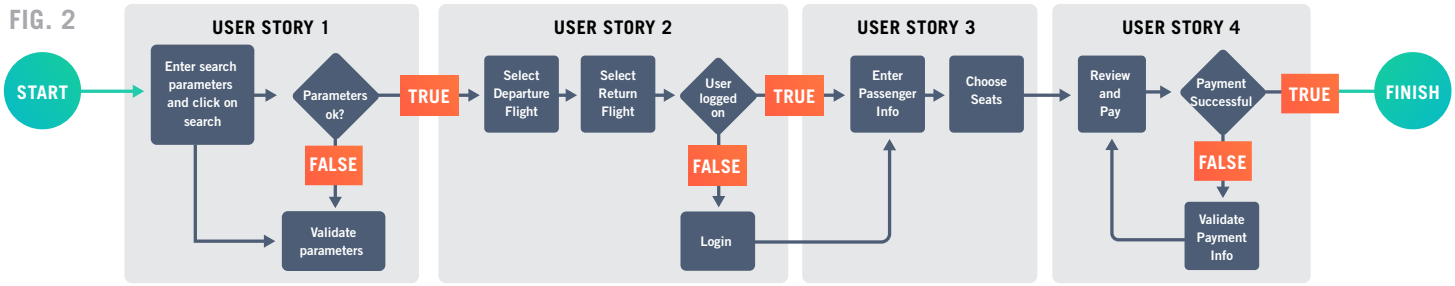
BUILDING THINGS RIGHT VS. BUILDING THE RIGHT THINGS

The use of diagrams such as visual flowcharts to represent requirements is something that helps analysts, product owners, developers, testers, and op engineers. Diagrams are a great communication tool to remove ambiguities and prevent misinterpretations by each of these stakeholders – ultimately leading to fewer defects in the code, as the visual flowcharts enable all stakeholders to have a common understanding from the get-go.

The key is to change our mindset of using "testing" as the only means to achieve application quality.

With Continuous Delivery we're realizing that although we can run unlimited automated tests at all levels to find defects, this approach will always be reactive and more costly than tests that always pass because there were no defects. That means we have **prevented**

FIG. 2



defects from being written into the code, which consequently means we have **built quality into the application** itself.

Martin Thompson says it best:
"It took us centuries to reach our current capabilities in civil engineering, so we should not be surprised by our current struggles with software."

We are on the right track. Tools have evolved and continue to evolve at a never-before-seen pace. The area that has been lagging in terms of advanced and easy-to-use tooling is the requirements-gathering and definition process. Martin Thompson also has a good quote on that:

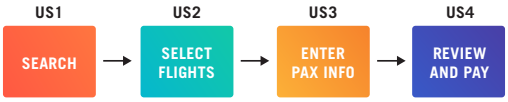
"If we look to other engineering [disciplines], we can see the use of tooling to support the process of delivery rather than imposing a process on it."

Look at civil engineering. CAD (computer-aided design) software revolutionized the designing of buildings and structures. We've been missing a CAD-like tool for software engineering, but now we are at a point where we have highly advanced and easy-to-use solutions to fill that gap.

BUILDING QUALITY INTO THE CODE = APPLICATION QUALITY

It is very common today for a product owner to draw an initial sketch on a whiteboard describing what she wants built. That sketch is then further refined through multiple iterations until the product owner is satisfied and **accepts** it.

That initial sketch for a simple Flight Booking Path example could look something like this:



Then, through multiple conversations with the product owner, developers, testers, and other stakeholders, the person assigned to formally model the Epic could come up with the following model shown at the top of the page.

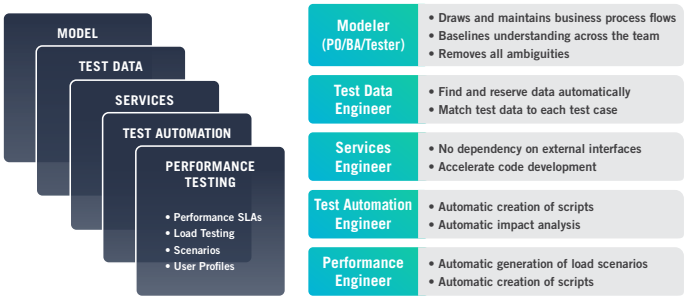
As you can see, those conversations caused a few additional process steps to be added as the model was formalized. We now know that the product owner wants the user to select the departure flight first and then select the return flight. It is also clear that before going to the passenger information step, the user must be prompted to log in. Lastly, it was clarified that the seats must be chosen only after the passenger information has been entered in the application.

Through the mere representation of the Epic in a visual model, ambiguities are removed and defects are prevented from entering the application code. Which means testing is truly "shifting left" in the lifecycle. And we're already starting to "build quality in" the application.

The visual model of the Flight Booking Path Epic becomes the foundational layer for other stakeholders in the lifecycle.

A CAD-like tool in software engineering helps us build a multilayered visual model of the requirements. These layers are tied together, and just like the CAD tools in civil engineering, the tool maintains full traceability across all layers as shown below.

LAYERED APPROACH TO CONTINUOUS DELIVERY



So if there is a change to any of those layers, the impact is automatically identified and communicated to the owner of each impacted layer, prompting the owner for a decision to address that impact.

From that visual model, the tool can then automatically:

1. Generate manual test cases.
2. Find, copy, mask, or synthetically generate the test data required for each test case.
3. Generate request/response pairs as well as provision virtual services for test cases to be able to run.
4. Generate test automation scripts in any language according to the test automation tools being used by the team.

So while developers must continue to invest in increasing code quality to **build things right**, the use of a CAD-like tool in software engineering not only accelerates the software lifecycle (i.e., speed), but it also ensures developers are **building the right things** (i.e., quality) from the beginning by providing unambiguous requirements to all stakeholders across the SDLC.

ALEX MARTINS has more than 18 years of experience in largescale application design, development and testing. For the last 13 years Alex has been focused on software quality engineering and testing discipline as the pillars for DevOps transformations. Going through all levels, from Tester to Practice Leader in various technology companies such as EDS, IBM, HP and Cognizant Technology Solutions, Alex built and ran several Enterprise Testing Organizations in Latin America and the US for multiple clients. Alex now works as a client advisor in the Continuous Delivery BU at CA Technologies and is also responsible for the Continuous Quality Center of Excellence. When not talking tech, you will either find Alex enjoying time with his family or on a beach somewhere surfing or kitesurfing.

cloudbees

THE HUB OF ENTERPRISE JENKINS AND DEVOPS

Software at the speed of ideas.

Continually build, deliver and improve the software that fuels your business with the CloudBees Jenkins Platform.

Your business has amazing, world-changing ideas – the only thing standing in your way is the time, energy and hassle it takes to turn code into finished product, to transform ideas into impact. CloudBees – the only secure, scalable and supported Jenkins-based DevOps platform – lets you focus on the ideas you want to bring to life, not the hassle of building, testing and deploying them, so you can start making an impact sooner.

CloudBees Jenkins Enterprise:
the enterprise Jenkins standard

CloudBees Jenkins Platform:
providing scalability, manageability, security, resililency

CloudBees Jenkins Platform – Private SaaS Edition:
adding elastic cloud capabilities

Try the CloudBees Jenkins Platform for free: www.cloudbees.com/get-started

Why You *Must* Care about DevOps

DevOps is becoming the de facto standard for software development. Companies that have adopted DevOps principles are disrupting industries, innovating faster, and leaving competitors behind. These companies have aligned internal stakeholders around the common objective of delivering quality software rapidly, frequently, and reliably.

Yet, despite the benefits DevOps provides, many organizations are reluctant to embrace it. Several factors underpin this reluctance, including resistance to change. There is also widespread misunderstanding of what DevOps is—a misunderstanding amplified by vendors who say DevOps is all about tooling.

There are also organizations that dismiss DevOps as a passing fad, or simply don't think that DevOps applies to them because they are not software companies. The facts, however, support the growing consensus that DevOps is here to stay, and that the benefits of DevOps extend to organizations in any industry. An increasing number of organizations are realizing that today they are a software company, that talented developers want to work for companies where they can innovate instead of fight fires all day, and that DevOps is a common-sense way to gain a sustainable, competitive business advantage.

Companies that have adopted DevOps principles are disrupting industries, innovating faster, and leaving competitors behind.

Any company that needs to deliver quality software faster needs to care about DevOps and the supporting practice of continuous delivery (CD), which enable continuously building, testing, and deploying software in frequent, incremental releases.

The rapidly mounting evidence strongly suggests that organizations that transition to DevOps get new innovations to market more quickly and sustain or gain competitive advantage.

WRITTEN BY BRIAN DAWSON
DEVOPS EVANGELIST AT **CLOUDBEES**

PARTNER SPOTLIGHT

CloudBees Jenkins Platform By CloudBees, Inc.

"With the CloudBees Jenkins Platform we are delivering more complex, larger projects more quickly." - **ADAM RATES,** HEAD OF STRATEGY AND ARCHITECTURE, ALLIANZ INSURANCE

CATEGORY
Continuous Deployment and CI Platform

NEW RELEASES
Continuously

OPEN SOURCE
No

CASE STUDY

Challenge: Allianz needed to respond to changes in the insurance market faster by shortening software delivery schedules.

Solution: Standardize on CI and CD with the CloudBees Jenkins Platform to minimize project startup times, automate development and testing tasks, and complete large, complex projects faster.

Industry Benefits: Project startup times were cut from days to minutes; reliable development and build environments were established; and staffing flexibility and scalability improved.

STRENGTHS

- Proven CI/CD platform
- Built-in high availability
- Role-based access control
- Advanced analytics
- Enterprise scale-out features

NOTABLE CUSTOMERS

- Allianz
- Netflix
- Adobe
- Thomson Reuters
- Hyatt Corporation
- Mozilla Corporation

BLOG cloudbees.com/blog | **TWITTER** @cloudbees | **WEBSITE** cloudbees.com

Autonomy or Authority: Why Microservices Should Be Event Driven

QUICK VIEW

01 Some of the recommendations we hear for building a microservices architecture goes against our sensibilities.

02 In this article we explore a different model to help guide us on our microservices journey and see why autonomy, not authority, is a key piece to the puzzle.

BY **CHRISTIAN POSTA**
PRINCIPAL ARCHITECT AT **RED HAT**

We've been discussing microservices in the mainstream for over two years now. We've heard many a blog or conference talk about what microservices are, what benefits they bring, and the success stories for some of the companies that have been at the vanguard of this movement. Each of those success stories center around the capacity for these companies to innovate, disrupt, and use software systems to bring value to their respective customers. Each story is slightly different but some common themes emerge:

- Microservices helped them scale their organization.
- Microservices allowed them to move faster to try new things (experiment).
- The cloud allowed them to keep the cost of these experiments down.

The question then becomes, "how does one get their hands on some microservices?"

We're told our microservices should be independently deployable. They should be autonomous. They should have explicit boundaries. They should have their own databases. They should communicate over lightweight transports like HTTP. Nevertheless, these things don't seem to fit our mental model very well. When we hear "have their own databases," this shatters our comfortable safety guarantees we know and love. No matter how many times we hear it, or even if we carved these postulates into stone, they're not going to be any more helpful. What might be helpful is trying to understand some of this from a different mental model. Let's explore.

In many ways, IT and other parts of the business have been built for efficiencies and cost savings. Scientific management has been the prevailing management wisdom for decades and our organizations closely reflect this (see Conway's law). To use a metaphor as a mental model: our organizations have been built like machines. We've removed the need for anyone to know anything about the purpose of the organization or their specific roles within that purpose and have built processes, bureaucracies, and punishments to keep things in order. We've squeezed away every bit of inefficiency and removed all variability in the name of repeatable processes and known outcomes. We are able to take the sum of the parts and do the one thing for which the machine was designed.

How does all this translate to our technology implementations? We see our teams organized into silos of development, testing, database administration, security, etc. Making changes to this machine requires careful coordination, months of planning and meetings. We attempt to change the machine while it's running full steam ahead. Our distributed systems reflect our organization: we have layers that correspond to the way we work together. We have the UI layer. The process management layer. The middleware, data access, and data layers. We've even been told to squeeze variability and

variety out of our services. Highly normalized databases. Anything that looks like duplication should be avoided. Build everything for reuse so we can drive costs down.

I'm not advocating for doing the opposite per-se, but there are problems with this model in our new, fast-changing competitive landscape. The model of a machine works great if we're building physical products: a car, a phone, or a microwave. If we already know what the outcome should be, this model has shown to be amazingly successful (see industrial revolution). However, this has existentially changed. Companies are building value these days through services, not through product alone. Service design is key. This model of a purpose-built machine is not a good model for building a service organization. Let's look at a different model.

What we really want is to find new ways to bring value to our customers through service and stay ahead of our competitors. To do that, we need to listen to our customers. To be able to react and fulfill their needs, we need to deal with the fact that customers don't know what they want. We need to explicitly deal with variety (law of requisite variety) and variability. We need to build "slack" into our systems to account for the "I don't know" factor. In many ways, innovation is about admitting "I don't know" and continually figuring out the right set of experiments to ask the right questions and then learn from the outcomes. Since software has eaten the world, this translates into building systems that have variability, feedback loops, and speed of change built into them. This looks very different from a machine. The model I like to use is a city.

Cities have many types of systems that co-exist, co-evolve, and exhibit a lot of the same behaviors we want. Emergent innovation through experimentation. There isn't top-down central control and planning and highly optimized silos. We have lots of independent, autonomous "agents" (people, families, businesses, etc.) and the operating environment (laws, physical geography, weather, and basic city services like roads, power, water, waste disposal, etc.). These agents interact in cooperative and also competitive ways. They interact with each other by asking each other to do things or responding to events to which they're exposed. These agents are driven by purpose (survival, personal/spiritual/monetary fulfillment, curiosity, etc.), and what emerges through these simple elements is an amazingly rich, resilient, and innovative ecosystem. Cities scale amazingly (see NYC). They innovate (see San Francisco, Seattle, etc.). There are no single points of failure. They recover from catastrophic failures (see natural or human-made catastrophes). And out of all of this, there is no single authoritative figure or set of figures that dictate how all of this happens.

This model of a city fits our microservices description a little better. Now let's start to translate to distributed systems. First, each agent (service) has its own understanding of the world. It has a history (series of events) that gives its current frame of reference from which it makes decisions. In a service, this frame of reference is implemented in its own database potentially. Services interact by asking each other to do something (commands) or responding to some given fact (events). It may observe an event and update its current understanding of the world (its history or its state). In this world, time and unreliability should be modeled explicitly.

A good way to do this is through passing messages. Things may not show up on time. Things might get lost. You may not be available to take commands. Other services may not be around to help you. You may favor autonomy (doing things yourself with the state you have) versus authority (asking someone else who may have the authoritative answer). To implement this, you may have to distribute knowledge about events and state through replication (in cities, we do this with broadcast mechanisms like newspapers, the internet, social networks, etc.).

When you do this, you may have to consider different state consistency models. Starbucks doesn't do two-phase commits. Not everything can/should expect a two-phase commit, consensus-approved consistency. You may need something more relaxed like sequential/causal/eventual consistency. You've got to make decisions with the state you have and potentially be in charge or resolving conflicts. You've got to equip your service to deal with these scenarios. If you cannot deal with them you need to learn from them. This requires changing behavior. In a microservices world this translates to quickly making changes to your service and getting them back out there via a CI/CD pipeline.

Microservices architectures are a type of complex-adaptive system. We need new models to reason about this. Thankfully a lot of the distributed system theory, research, and practices have been around for 40+ years and innately take this type of system into account (as a distributed system is itself a complex-adaptive system!). We don't have to reinvent the wheel to do microservices, just re-orient our mental model and be a catalyst for change throughout our organizations.

CHRISTIAN POSTA (@christianposta) is a Principal Architect at Red Hat and well known for being an author (Microservices for Java Developers, O'Reilly 2016), frequent blogger, speaker, open-source enthusiast, and committer on Apache ActiveMQ, Apache Camel, Fabric8, and others. Christian has spent time at web-scale companies and now helps companies creating and deploying large-scale distributed architectures - many of what are now called Microservices based. He enjoys mentoring, training, and leading teams to be successful with distributed systems concepts, microservices, DevOps, and cloud-native application design.

Test and deploy your code with confidence.

Easy Setup

Login with GitHub, tell Travis CI to test a project, and then push to GitHub. Could it be any simpler!

Test Pull Requests

Make sure every pull request to your project is tested before merging.

Customizable

Quickly configure your repository settings so your builds will run the way you need them to.

Travis CI

travis-ci.com

Microservices Are a Keystone of Continuous Delivery

Some consider microservices a technology du jour, another fad that's due to pass; others (including myself) argue that instead they're the missing piece for truly enabling teams to adopt Continuous Delivery.

HIGHER VELOCITY

With microservices focusing on small pieces of functionality of a larger application, it's become much easier to iterate around single pieces, getting new features and bug fixes in front of customers faster, shortening the feedback cycle significantly.

FEWER DEPENDENCIES

Smaller services have a key benefit that primes them for Continuous Delivery. Their sharp focus on a specific application concern keeps their dependencies small, isolating them from variances and changes while making it easier to standardize on build environments. Most importantly, fewer dependencies help keep the build fast.

MANAGING COMPLEXITY

In return, communication between teams working on different microservices has become more important than ever. An increasing number of services talking to each other consequently increases the complexity in interactions, both between the services themselves and the teams working on them.

WORKING AT (TEAM) SCALE

Microservices scale well across teams, allowing new services and teams to come and go as needed. In turn, an organization needs a delivery tool that can adopt to those ever-changing needs quickly and seamlessly.

Microservices enable the fast feedback cycles that Continuous Delivery promotes, but teams adopting them need to make sure that their tool of choice can support the benefits that microservices bring while also enabling them to scale up to new services and new teams easily.

WRITTEN BY MATHIAS MEYER
CO-FOUNDER AND CEO AT **TRAVIS CI**

PARTNER SPOTLIGHT

Travis CI By Travis CI

The continuous integration and delivery platform your developers love, trusted by hundreds of thousands of open-source projects, teams, and developers.

CATEGORY
Continuous Delivery and Integration

NEW RELEASES
Continuous

OPEN SOURCE
Mostly

TRAVIS CI ENTERPRISE

Travis CI Enterprise brings all the features that developers know and love, running on your own infrastructure.

Integrating deeply with GitHub Enterprise, it covers all the security and regulatory requirements, protecting your data, using your infrastructure and secure login via SAML or LDAP.

Running Travis CI Enterprise on your own infrastructure gives you full control over your build resources as well as the build environment. Adding more is as easy as provisioning a new machine with just a few commands that can be fully automated.

Whether you're using EC2, OpenStack, Azure, Google Compute Engine, or your own hardware, Travis CI runs anywhere.

Find out more: enterprise.travis-ci.com

STRENGTHS

- Easy configuration, stored and versioned alongside your project's code.
- Scales with your team, the workflow ready for companies large and small.
- No more "works on my machine" thanks to isolated build environments.
- Trusted by hundreds of thousands of open-source projects, developers, and teams.
- Seamless integration with GitHub, first-class support for pull request workflows.
- Available hosted, and on-premises for use with GitHub Enterprise.

NOTABLE CUSTOMERS

SAP	Zendesk	Fastly
NASDAQ	Airbnb	

BLOG blog.travis-ci.com

TWITTER @travisci

WEBSITE travis-ci.com

Executive Insights on the State of DevOps

BY **TOM SMITH**
RESEARCH ANALYST AT **DZONE**

QUICK VIEW

01 The most important elements of DevOps are: 1) people, 2) process, and 3) technology.

02 The biggest change being seen in DevOps is its transition from a "new and different" way of working to "the new normal."

03 The greatest value of DevOps is improved speed to market with a quality product that meets or exceeds customer expectations.

To gather insights on the state of the DevOps movement in 2017, we talked to 16 executives from 14 companies who are implementing DevOps in their own organization and/or providing DevOps solutions to other organizations. Specifically, we spoke to:

MICHAEL SCHMIDT, Senior Director, Automic
AMIT ASHBEL, Director of Product Marketing & Cyber Security Evangelist, Checkmarx
SACHA LABOUREY, CEO and Founder, CloudBees
SAMER FALLOUH, V.P. Engineering and
ANDREW TURNER, Senior Solution Engineer, Dialexa
ANDREAS GRABNER, Technology Strategist, Dynatrace
ANDERS WALLGREN, CTO, Electric Cloud
JOB VAN DER VOORT, V.P. of Product, GitLab
CHARLES KENDRICK, CTO, Isomorphic Software
CRAIG LUREY, CTO and Co-Founder, Keeper Security
JOSH ATWELL, Developer Advocate, NetApp SolidFire
JOAN WRABETZ, CTO, Quali
JOE ALFARO, V.P. of Engineering, Sauce Labs
NIKHIL KAUL, Product Marketing Manager Testing and
HARSH UPRETI, Product Marketing Manager API, SmartBear Software
ANDI MANN, Chief Technology Advocate, splunk

KEY FINDINGS

01 The most important elements of DevOps are **1) people; 2) process; and, 3) technology.** People are the most important since they need to change the culture and the mindset. Process involves tearing down walls between all departments, defining the process, and creating controls that cannot be violated – by anyone. Optimize the use of third-party technology to automate everything you possibly can. This is necessary to be able to scale and to make the feedback loop smaller, to release faster, to increase quality, and to reuse assets across development, testing, and monitoring. While these changes are not easy, they will ultimately improve the quality of life for everyone in development, operations, security, testing, and sales, as well as the end-user experience (UX).

02 The biggest change to DevOps has been its **acceptance**. DevOps has gone from being new and different to the "new normal." DevOps outperforms any other method by a significant margin, is causing significant disruption in the industry - even in enterprise companies with legacy systems that are willing to make the change. It has moved from just being a bottom-up implementation to top-down since senior management can see the results it achieves.

The next biggest changes identified are the increase in automation and collaboration. More automation and less operations solves problems faster. Automation is built in up front to optimize build time versus set-up and onboarding. More focus on containers and replicable builds assist with scaling, sharing, and collaboration.

We're not only breaking down walls between developers and operations but also with support, sales, and marketing to meet the needs of all departments. It's still about people collaborating and finding a better way to get something developed.

03 The greatest value delivered by DevOps is **improved speed to market with a quality product that meets or exceeds customer expectations**. Fast iteration gets new features out faster at higher quality, delivering value more quickly. You're faster because you're focusing on the business value and building what the customer wants rather than some shiny new technology. Continuous deployment enables features and fixes to be rolled out more quickly and enhances customer engagement. This reduces

wasteful development. We've seen a 30% improvement in customer satisfaction, a 10x increase in innovation velocity, and a 2x increase in efficiency.

DevOps is a repeatable process where every member of the team knows where everything is in the process, and each team member knows their role. This results in a more reliable path, faster and more stable releases. Better teamwork equals better products.

04 Most respondents are **using their own solutions to implement DevOps**. In addition, the most frequently used tools are Confluence, Jenkins, JIRA, Team City, and Travis, with 14 others being mentioned.

05 Real-world problems being solved by DevOps revolve around **improving quality, speed to market, and reducing cycle times with automation**. An online trading company went from deployments that took weeks to deployments in under a minute. The ability for any company to test on all operating systems, browsers, and devices, thereby removing testing from the chain of development, lets developers focus on building business value. DevOps provides the ability to receive a bug report and turn around a fix the same day with confidence the fix has not introduced any subtle regressions. Big companies with geographically distributed developers have the ability to check in source code to expedite the development process. Replacing manual testing with automated can save hundreds of thousands of dollars or reduce headcount by 35%. A gaming company with a complex application portfolio can make several thousand updates every day, while a telecommunications company with more than 100 apps can combine their business and operations systems on the same platform.

06 The **obstacles to success revolve around people**. People in power can be slow to change. Kingdoms may go away, and this results in power battles. People in management are too comfortable in their current jobs and don't want to change. There's a fear of having greater responsibility for the code running in production. However, this gives the engineering team a lot of freedom to try new things and deploy as they wish with almost instant gratification by seeing the impact code has on end users. Everything is mental – the "not built here mentality," especially with start-ups. Remove this mentality and focus on where your company adds value. Don't worry about your kingdom, worry about the needs of your customers.

DevOps is a process of incremental improvement that takes time. It's an evolutionary method, not a quick fix. Netflix and Google took seven years. It may be painful getting started, but once you get past the initial start-up issues and address them, the tools and the automation reduce the number of incidents, and, consequently, the amount of pain felt by the organization. Resist the urge to go backwards in the process for sake of urgency. Do the hard work, put the processes, tools, and automation in place, and you will reap the benefits.

07 The greatest concern regarding DevOps goes back to **people not being willing to change or not being fully committed to the change**. You must change to produce value and velocity. If you don't change, you'll lose people and be replaced by your competitors who have embraced change to develop quality code more quickly.

Early in the adoption and implementation, people will blame tools for finding the errors in the code. However, root-cause analysis will help uncover the source of the problem.

Help people collaborate, communicate, and integrate to deliver better software. Build security and testing into automated processes from the beginning and see the benefits of the journey rather than looking for a perfect end state (hint: it doesn't exist).

08 The greatest opportunities for the future of DevOps are: **1) continued growth of the cloud; 2) tools facilitating automation; 3) containers; and 4) company-wide collaboration**. There's a healthy symbiosis between DevOps and the cloud. As AWS, Red Hat, and Azure grow, DevOps will grow exponentially. AWS is providing more tools to facilitate automation, and there is consolidation of tools so you don't have to custom build deployments. We will see greater adoption of the public cloud by regulated industry. Containers will become more important automating the flow through the cloud. DevOps will affect all aspects of the organization fostering communication and collaboration among developers, operations, QA, security, testing, deployment, business planning, and BPO to provide a better user experience.

There's only a two percent adoption of DevOps practices around the world. We will move from early adoption to general adoption by sharing stories about what we've done and how tools can remove fear and pain with ongoing monitoring.

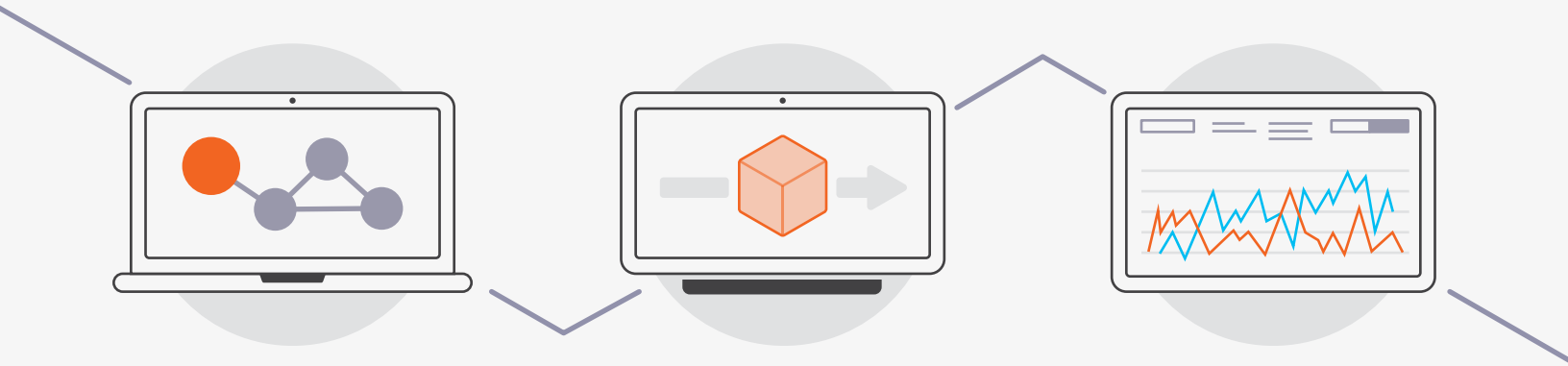
09 One consistent theme about what developers need to keep in mind with regards to DevOps and Continuous Delivery is the need to **be flexible and collaborative**. Be open to collaboration, sharing, and gaining empathy for those you are working with – operations, as well as the end user. In addition, don't boil the ocean. Find a problem and fix a problem. Fail fast, cheap, and small. Don't be tempted to move backward in the DevOps process. Everything has a tax associated with it. Keep this in mind and commit to the DevOps process. The more you know about testing, writing code for testing, and writing automation, the more valuable you will be as DevOps becomes more ingrained.

10 A couple of themes surfaced when we asked respondents what else we needed to consider with regards to DevOps for this research guide. We learned it's important to understand that **DevOps is not a product, it's an implementation of a manufacturing process that relates to technology**. DevOps principles are applicable throughout the industry, reducing uncertainly and improving the efficiency of process management. Furthermore, there is the need for **operations and developers to be using the same set of tools to solve common problems** whereby the tools are facilitating communication, collaboration, and problem solving.

TOM SMITH is a Research Analyst at DZone who excels at gathering insights from analytics—both quantitative and qualitative—to drive business results. His passion is sharing information of value to help people succeed. In his spare time, you can find him either eating at Chipotle or working out at the gym.

weaveworks

HELPING DEV AND DEV/OPS OPERATIONALIZE CLOUD NATIVE APPS IN CONTAINERS



Connect, **Deploy**, **Monitor**, and **Troubleshoot** Microservices and Containers

CONTAINER NETWORKS & FIREWALLS

Enable automatic service discovery, firewalling and segmentation for your microservices. Weave provides resilient container networking, security, and network segmentation.

CONTINUOUS DELIVERY

Manage and control when and how changes get deployed to different environments. Weave Cloud makes Continuous Delivery as simple as a `git push`.

PROMETHEUS MONITORING

Focus on creating code while Weave Cloud provides a hosted, horizontally scalable Prometheus monitoring service.

TROUBLESHOOTING DASHBOARD

Deploy your app and verify it in Weave Cloud. See the network, the hosts, and application to discover and resolve problems quickly to improve Mean Time to Recovery..

TRY WEAVE CLOUD FREE FOR 60 DAYS

https://www.weave.works/cloud

https://www.weave.works/blog/ https://www.facebook.com/WeaveworksInc https://twitter.com/weaveworks

Ship features & fix issues faster with Continuous Delivery to Kubernetes

Achieving continuous delivery improves deployment time and reliability. Yet only 40% of developers doing CI are doing CD. With Weave Flux, adding CD to your CI pipeline is simple. Kubernetes is a powerful platform for running Docker containers. You'll write a "manifest," which is a YAML file that describes which containers to run as part of the app.

An example manifest (from https://git.io/vMd2M) looks like:

```
containers:
- name: front-end
  image: weaveworksdemos/front-end:0.3.1
```

The version tag is '0.3.1', the best practice being that you version control your manifests and tag all of your Docker images. It's bad news if you use 'latest' and then can't control which version of your Docker image is running in production!

Automatically enable continuous delivery to Kubernetes while maintaining best practice of storing config in version control.

But how can you automatically update the version tag in your manifest?

You could write a script to clone your config repo, parse the YAML, automatically replace the version string in your manifests, push the change to your config repo, and deploy it to Kubernetes. But that's a lot of work—and then you have to maintain it. Then what happens if you want to rollback a release to a previous version? Or have different policies for prod and staging?

Weave Flux solves these problems. Weave Flux is an open-source project, and available as part of the Weave Cloud service. Weave Cloud also provides tools to operate containers such as Prometheus monitoring, real-time observation, and a troubleshooting dashboard.

Join our user group and attend one of our free online trainings about Kubernetes, Weave Cloud, and more here.

WRITTEN BY LUKE MARSDEN
HEAD OF DEVELOPER EXPERIENCE AT **WEAVEWORKS**

PARTNER SPOTLIGHT

Weave Cloud By Weaveworks

weaveworks

Weave Cloud makes it simple for DevOps team to operationalize containers, while using their choice of orchestration platform

CATEGORY
Containers and Microservices / SaaS

NEW RELEASES
Continuously Updated

OPEN SOURCE
Yes

CASE STUDY

Adopting containers and microservices can be challenging. Big platforms add complexity that give development teams headaches. Convenient, flexible, and open source, Weave Cloud simplifies delivery for cloud-native development.

Weave Cloud is a SaaS offering that lets Dev/DevOps operationalize containers via extensions to their choice of orchestration platform. Weave Cloud accelerates development, replacing manual configuration and scripting with automation, and provides all of the container and orchestrator-level information needed to manage. It delivers features like Prometheus monitoring, continuous deployment, firewall management, and troubleshooting.

STRENGTHS

- Deploy, troubleshoot, monitor, and secure containers across hosts.
- Setup security policy and firewalls.
- Troubleshoot and explore your app topology in real-time.
- Deploy and upgrade apps safely to the cloud continuously.
- Monitor your app and infrastructure with Prometheus monitoring.

SUPPORTED INTEGRATIONS

- Kubernetes
- Docker
- AWS
- Microsoft Azure
- Mesosphere

BLOG weave.works/blog | **TWITTER** @weaveworks | **WEBSITE** weave.works

Branching Considered Harmful!

(for Continuous Integration)

BY **ANDREW PHILLIPS**
VP DEVOPS STRATEGY AT **XEBIALABS**

QUICK VIEW

01 Branch-based development can be a good trade-off in scenarios such as open-source projects, but should be Considered Harmful for genuine Continuous Integration.

02 A trunk-based approach can work, and it enforces good practices we otherwise often fail to get around to.

03 Branches can also be useful in a trunk-based development environment.

04 Consider trunk-based development as a goal for your teams.

If you're working in what you might consider a reasonably modern software development environment, chances are you're doing Agile, and are aspiring to build out a Continuous Integration and Deployment pipeline to push code to production quickly. Chances are that you're also doing some form of branch-based development – whether straightforward feature branches, something more formalized such as git-flow, or a pull request-based process using GitHub or similar.

Why the branches? How and when did the perception arise that branching is A Good Thing? A branch-based development strategy generally implies delayed merging, and delayed merging is, almost literally, the exact opposite of the "continuous integration" most teams consider themselves to be practicing. Branch-based development usually means either largely more complex and risky merges, or a manual code review bottleneck, or both – things we almost never want in our process if the aim to deliver code to production quickly.

I should briefly clarify that what I mean by "branch-based development strategy" here are approaches that allow for, or even encourage, the existence of branches that accumulate changes that are not on trunk (or "master", or "HEAD", or whatever the mainline branch in your environment is called) for days, weeks, or longer. These changes are only incorporated into trunk after an explicit, non-automated approval. There are also development strategies using branches that in practice look much more like trunk-based development – more on those later.

But isn't branch-based development so well supported by leading source control management systems such as Git because it's A Good Thing? Aren't contributions via pull requests, and hence branches, the way many, many well-known and successful open-source projects work?

Yes, many open-source projects, including the Linux kernel community that gave rise to Git, use branches in their development strategy – and for good reason. If you are working in a highly distributed environment where there is no centralized server, there can be no notion of "merging to master" or "committing to trunk", because no such thing as trunk exists. Determining the "current global state of trunk" requires merging multiple tracks of development from separate repositories.

Even if you have a centralized server which is the canonical reference for a particular codebase, branches and pull request reviews make sense if you are in an environment where many contributions are being proposed by developers who are not familiar with the code, and who in addition are not in a position to ask for advice from more expert developers on an ongoing basis. They may be geographically separated, with few or no overlapping working hours to allow for discussions, or they may work for different companies and simply do not have the opportunity to contact the experts often.

In such cases, a development strategy using branches and delayed integration – usually after manual code review – makes sense as an acceptable compromise given the various constraints. However, projects in these situations are very rarely examples of code that gets shipped to production frequently. So if that is our goal, they hardly seem like models that we should be seeking to emulate.

More importantly, the constraints under which these projects operate are simply not applicable to most commercial software development settings. We almost always have a central repository whose mainline is the global reference point so that

the idea of "merging to trunk" on an ongoing basis is indeed meaningful. And we usually have sufficiently frequent access to someone with enough experience of the codebase that we could arrange for reasonably regular feedback if we so chose.

"So what exactly are you proposing?" I hear you ask. I am proposing something we'll refer to here as "trunk-based development." It's not a new concept by any means, having been around at least as long as the idea of continuous integration, to which it is closely related. But I think that, if we are serious in our desire to ship code quickly, it's definitely a concept worth revisiting. It goes something like this:

1. When you have written an amount of code locally that you think should work, commit it to trunk.
2. A commit to trunk triggers a series of tests and verification steps that are intended to ensure that the system will not be negatively impacted by the change.
3. If all the tests and verification steps succeed, the code change is deployed to production.
4. Otherwise, the commit could be rolled back automatically, or the "offending" developer is notified that they have broken trunk.

That's all – a pretty straightforward setup. But what about style checks or architectural reviews? What about all those work-in-progress commits? Won't we be breaking production all the time with problems the tests don't catch, or with incompatibilities that we're introducing?

Yes, in most current setups committing straight to trunk and thence to production would indeed end up breaking things a fair amount of the time. And that is precisely the point: by allowing ourselves to develop "off trunk" for long periods of time, we allow ourselves to ignore the work necessary to *protect* trunk and production from bad code.

Avoiding work may not sound like a bad thing, but it almost always comes at the cost of a steadily worsening review bottleneck that can see even simple changes delayed for days or more. And if we are truly aiming for a continuous deployment setup, we will eventually need to put in the effort to protect trunk *anyway*.

Furthermore, the steps we need to take in a trunk-based development environment in order to prevent production incidents are exactly the kinds of best practice steps we generally talk about striving for.

Want more architectural and code style input? Introduce regular "check-ins" with a more experienced colleague, or do more pair programming! Concerned about checking in code that compiles but doesn't work correctly when it makes it to production? Write more tests to ensure your code does what you expect! Working on a feature that shouldn't be active until changes to other systems are deployed? Use feature flags! Aware that your system is too complicated to be tested reliably with unit and integration tests? Implement automated monitoring of system health, customer metrics, etc., and alert or roll back if behavior is unexpectedly poor! And so on.

These are all things we know about and generally agree that we should have – we just never get round to them because we allow ourselves to work in a manner that doesn't *compel* us to implement them to keep our systems running.

But if we *do* force ourselves to go down this path, the benefits in terms of speed can be enormous. A colleague who tried to see how far he could take this ended up with what he called "ludicrous mode": every time a file was saved in his IDE, the code would be compiled and, if successful, the unit tests for that project were run. If those succeeded, the change would automatically be committed to trunk and, if no stages in the subsequent delivery pipeline failed, would make its way all the way to production.

At his peak, he was deploying to production *fifty* times an hour. But he said that the first thing the experiment taught him is that it was essential to write failing unit tests *first* before starting on the actual implementation, because that was required to ensure bad code didn't make it to production. Good practice not as a nice-to-have, but out of necessity!

Having said all that, in practical terms there are still good use cases for branches, of course: for example, they are often the easiest way of "throwing a code diff out there," especially during a discussion with remote participants. It's also easier in many current continuous integration and deployment tools to configure a "create pull request, trigger tests, automatically merge to trunk, continue pipeline" flow than the "merge to trunk, trigger tests, continue pipeline or automatically revert" process sketched above.

The key to both of these cases, though, is that they involve, respectively, throw-away and short-lived branches that are *automatically* merged into trunk. There is no long-running "off trunk" development followed by a large merge or manual review.

Of course, for most teams (including the ones I am part of!) trunk-based development isn't something that could be implemented today: a lot of the work required to protect trunk from bad commits is still needed, and that work isn't easily or quickly done. But we can embrace trunk-based development as a desirable goal and reconsider our use of branch-based approaches that build delay and manual bottlenecks into our flow to production. That way, we can get back to focusing on what it would take to build a system and process that will allow us to deliver code as quickly and safely as we can.

So go on, ask yourself: what would *your* team need to do to be able to develop safely on trunk?

ANDREW PHILLIPS heads up strategy at XebiaLabs, developing software for visibility, automation, and control of Continuous Delivery and DevOps in the enterprise. He is an evangelist and thought leader in the DevOps and Continuous Delivery space. When not "developing in PowerPoint," Andrew contributes to a number of open-source projects, including the multi-cloud toolkit Apache jclouds.

A brief history of web and mobile app testing.

Find out how Sauce Labs can accelerate your testing to the speed of awesome.

For a demo, please visit saucelabs.com/demo
Email sales@saucelabs.com or call (855) 677-0011 to learn more.

SAUCELABS
Testing at the speed of awesome.

Automated Testing: The Glue Between Dev and Ops

In order to reap the full benefits of DevOps, organizations must integrate automated software testing into their continuous delivery pipelines. It is the only way to ensure that release occurs at both a high frequency, and with a high level of quality.

In order to integrate continuous testing effectively into a DevOps toolchain, look for the following essential features when evaluating an automated testing platform:

- **Support for a variety of languages, tools, and frameworks.** The programming languages and development tools that your DevOps team uses today are likely to change in the future. Look for a testing solution that can support a broad array of languages, tools, and frameworks.
- **Cloud testing.** On-demand cloud-based testing is the most cost-efficient option because it obviates the need to set up and maintain an on-premises test grid that is underutilized most of the time. It also reduces the resource drain associated with identifying and resolving false positives, or failures due to problems in the test infrastructure.
- **The ability to scale rapidly.** Your testing platform should be able to perform tests as quickly as needed, and be able to do so across all required platforms, browsers, and devices. It should also be highly scalable to support as many parallel tests at one time as you require.
- **Highly automated.** DevOps teams achieve their speed and agility in part by automating as much of the software delivery process as possible. Your testing solution should work seamlessly with other components of your toolchain, most notably your CI and collaboration tools.
- **Security.** In a DevOps environment, all members of the team have an important role to play in keeping applications secure. Testing platforms, therefore, need enterprise-grade security features.

A software testing platform that includes these qualities will empower your organization to derive full value from its migration to a DevOps-based workflow by maximizing the agility, scalability, and continuity of your software delivery pipeline.

WRITTEN BY LUBOS PAROBEK
VP OF PRODUCTS, **SAUCE LABS**

PARTNER SPOTLIGHT

Automated Testing Platform By Sauce Labs

SAUCELABS

Sauce Labs accelerates the software development process by providing the world's largest automated testing cloud for mobile and web applications.

CATEGORY
Automated Testing Platform

NEW RELEASES
Daily

OPEN SOURCE
Yes

CASE STUDY

As a cutting edge development team, Dollar Shave Club was quick to adopt DevOps. However, the QA team was spending too much time testing their website. In fact, it took more than 10 hours to do a full test suite, and it did not cover the diverse set of devices and browsers for their users. To increase its test coverage while reducing test execution time, Dollar Shave Club decided on Sauce Labs for cloud-based automated testing. The QA team now runs automated functional, regression, and configuration tests in parallel for both its web and mobile applications. Each day, Dollar Shave Club can now run more than one hundred unique tests on desktop web, mobile web, and its native iOS and Android apps. Each test runs several times a day, on multiple browsers and platforms. Because Sauce Labs integrates with their Jenkins Continuous Integration (CI) server, the company can test code automatically with each commit, getting results back within 10 minutes of each check-in. By running their tests in parallel on Sauce Labs, Dollar Shave Club is saving hundreds of thousands of dollars per year due to reduced testing time and increased developer productivity.

STRENGTHS

- Enterprise-grade cloud-based test infrastructure provides instant access to more than 800+ browser/OS/platform configurations
- Highly scalable, on-demand platform reduces testing time from hours to minutes when tests are run in parallel
- Optimized for CI/CD workflows, testing frameworks, tools, and services
- Single platform for all your testing needs—automated & manual desktop web, mobile web, and native/hybrid iOS and Android apps

NOTABLE CUSTOMERS

- American Express
- IBM
- Salesforce
- Home Depot
- New Relic
- Slack

BLOG saucelabs.com/blog

TWITTER @saucelabs

WEBSITE saucelabs.com

Microservices and Docker at Scale:

The PB&J of Modern Application Delivery

BY **ANDERS WALLGREN**
CTO AT **ELECTRIC CLOUD**

QUICK VIEW

01 Microservices and containers have several benefits, including: allowing organizations to "divide and conquer," scale teams and applications more efficiently, and offer a consistent, isolated runtime environment.

02 Microservices and containers are complex patterns/tools for solving complex problems. They are not for everyone. Key challenges are around test automation, pipeline variations, operations complexity, monitoring, logging, and remediation.

03 You need to figure out if they are right for your use case, and ensure you develop the skills and key prerequisites to make microservices and large-scale container deployments work for you.

Microservices and containers have recently garnered a lot of attention in the DevOps community. Docker has matured, and is expanding from being predominantly used in the Build/Test stages to Production deployments. Similarly, microservices are expanding from being used mostly for greenfield web services to being used in the enterprise—as organizations explore ways to decompose their monolith applications to support faster release cycles.

As organizations strive to scale their application development and releases to achieve Continuous Delivery, microservices and containers, although challenging, are increasingly considered. While both offer benefits, they are not "one size fits all," and we see organizations still experimenting with these technologies and design patterns for their specific use cases and environment.

WHY MICROSERVICES? WHY CONTAINERS?

Microservices are an attractive DevOps pattern because of their enablement of speed to market. With each microservice being developed, deployed, and run independently (often using different languages, technology stacks, and tools), microservices allow organizations to "divide and conquer," and scale teams and applications more efficiently. When the pipeline is not locked into a monolithic configuration—of either toolset, component dependencies, release processes, or infrastructure—there is a unique ability to better scale development and operations. It also helps organizations easily determine what services don't need scaling in order to optimize resource utilization.

Containers offer a well defined, isolated runtime environment. Instead of shipping an artifact and all of its variables, containers support packaging everything into a Docker-type file that is promoted through the pipeline as a single container in a consistent environment. In addition to isolation and consistent environment, containers also have very low overhead of running a container process. This support for environment consistency from development to production, alongside extremely fast provisioning, spin-up, and scaling, accelerate and simplify both development and operations.

WHY RUN MICROSERVICES IN CONTAINERS?

Running microservices-based applications in a containerized environment makes a lot of sense. Docker and microservices are natural companions, forming the foundation for modern application delivery.

At a high level, microservices and Docker together are the PB&J of DevOps because:

- They are both aimed at doing one thing well, and those things are complimentary.
- What you need to learn to be good at one translates well to the other.

More specifically:

- **Purpose**
  - A microservice is (generally) a single process focused on one aspect of the application, operating in isolation as much as possible.
  - A Docker container runs a single process in a well-defined environment.
- **Complexity**
  - With microservices you now need to deploy, coordinate, and run multiple services (dozens to

hundreds), whereas before you might have had a more traditional three-tier/monolithic architecture. While microservices support agility—particularly on the development side—they come with many technical challenges, mainly on the operations side.

- Containers help with this complexity because they make it easy and fast to deploy services in containers, mainly for developers.

- **Scaling**
  - Microservices make it easier to scale because each service can scale independently of other services.
  - Container-native cluster orchestration tools, such as Kubernetes, and cloud environments, such as Amazon ECS and Google Container Engine (GKE), provide mechanisms for easily scaling containers based on load and business rules.

- **System Comprehension**
  - Both microservices and containers essentially force you into better system comprehension—you can't be successful with these technologies if you don't have a thorough understanding of your architecture, topology, functionality, operations, and performance.

Challenges

Managing microservices and large-scale Docker deployments pose unique challenges for enterprise IT. Because there is so much overlap in terms of what an organization has to be proficient at in order to successfully deploy and modify microservices and containers, there is quite a bit overlap in terms of challenges and best practices for operationalizing containers and microservices at scale.

- Increased pipeline variations: Orchestrating the delivery pipeline becomes more complex, with more moving parts. When you split a monolith into several microservices, the number of pipelines might jump from one to 50 (or however many microservices you have set up). You need to consider how many different teams you will need and whether you have enough people to support each microservice/pipeline.
- Testing becomes more complex. There is a larger amount of testing that needs to be taken into consideration–integration testing, API contract testing, static analysis, and more.
- Deployment complexity increases. While scaling the containerized app is fairly easy, there's a lot of activity that needs to happen first. It must be deployed for development and testing many times throughout the pipeline, before being released to production. With so many different services developed independently, the number of deployments increases dramatically.
- Monitoring, logging, and remediation become very important and increasingly difficult because there are more moving parts and different distributed services that comprise the entire user experience and application performance.
- There are numerous different toolchains, architectures, and environments to manage.
- There is a need to take into consideration existing legacy applications and how these are coordinated with the new services and functionality of container- or microservices-based applications.
- Governance and auditing (particularly at the enterprise level) become more complicated with such a large distributed environment, and with organizations having to support both containers and microservices, alongside traditional releases and monolithic applications.

In addition to these common challenges, microservices and containers each pose their own unique challenges. **If you're considering microservices, know that:**

- Distributed systems are difficult and mandate strong system comprehension.
- Service composition is tricky and can be expensive to change. Start as a monolith, and avoid premature decomposition, until you understand your application's behavior thoroughly.
- Inter-process failure modes need to be accounted for and although abstractions look good on paper they are prone to bottlenecks.
- Pay attention to transaction boundaries and foreign-key relationship as they'll make it harder to decompose.
- Consider event-based techniques to decrease coupling further.
- For API and services' SLA - *"Be conservative in what you do, be liberal in what you accept from others"*
- State management is hard–transactions, caching, and other fun things...
- Testing (particularly integration testing between services) and monitoring (because of the increased number of services) become way more complex.
- Service virtualization, service discovery, and proper design of API integration points and backwards-compatibility are a must.
- Troubleshooting failures: "every outage is a murder mystery."
- Even if a service is small, the deployment footprint must be taken into account.
- You rely on the network for everything-you need to consider bandwidth, latency, reliability.
- What do you do with legacy apps: Rewrite? Ignore? Hybrid?

For containers:

- Security is a critical challenge–both because it is still a relatively new technology, and due to the security concerns for downloading an image file. Containers are black boxes to OpSec: less control, less visibility inside the container, and existing tools may not be container-savvy. Be sure to sign & scan images, validate libraries, etc.; harden the container environment as well; drop privileges early, and use fine-grained access control so it's not all root. Be smart about credentials (container services can help).
- Monitoring is tricky, since container instances may be dropped or span-up continuously. Logging and monitoring needs to be configured to decommission expired containers, or save the log and data-business data, reference data, compliance data, logs, diagnostics-from other (temporal) instances.
- Know what's running where, and why, and avoid image bloat and container sprawl.
- Since the containers hosting and cluster orchestration market is still emerging, we see users experimenting a lot with running containers across multiple environments, or using different cluster orchestration tools and APIs. These early adopters need to manage containers while minimizing the risk of lock-in to a specific cloud vendor or point-tool, or having to invest a lot of work (and steep learning curve) in rewriting complex scripting in order to repurpose their deployment or release processes to fit a new container environment or tool.

BEST PRACTICES FOR MICROSERVICES AND CONTAINERS

While, admittedly, there are a fair number of challenges when it comes to deploying microservices and containers, the end-result, will be reduced overhead costs and faster time to market. If microservices and containers make the most sense for your application use case, there is a great deal of planning that needs to happen before you decompose your application to a set of hundreds of different services, or migrate your data center to a container environment. Without careful planning and following industry best practices, it can be easy to lose the advantages of microservices and containers.

To successfully run microservices and containers at scale, there are certain skill sets that the organization must possess throughout the software delivery cycle:

- **Build domain knowledge.** Before deploying microservices it is critically important to understand the domain before making difficult decisions about where to partition the problem into different services. Stay monolithic for a while. Keep it modular and write good code.
- Each service should have independent **CI and Deployment pipelines** so you can independently build, verify, and deploy each service without having to take into account the state of delivery for any other service.
- **Pipeline automation:** A ticketing system is not automation. With the increase in number of pipelines and pipeline complexity, you must be able to orchestrate your end-to-end process, including all the point-tools, environments, and configuration. You need to automate the entire process-from CI, testing, configuration, infrastructure provisioning, deployments, application release processes, and production feedback loops.
- **Test automation:** Without first setting up automated testing, microservices and containers will likely become a nightmare. An automated test framework will check that everything is ready to go at the end of the pipeline and boost confidence for production teams.
- **Use an enterprise registry for containers.** Know where data is going to be stored and pay attention to security by adding modular security tools into the software pipeline.
- Know what's running where and why. Understand the platform limitations and avoid image bloat.
- Your pipeline must **be tools/environment agnostic** so you can support each workflow and tool chain, no matter what they are, and so that you can easily port your processes between services and container environments.
- **Consistent logging and monitoring** across all services provides the feedback loop to your pipeline. Make sure your pipeline automation plugs into your monitoring so that alerts can trigger automatic processes such as rolling back a service, switching between blue/green deployments, scaling, and so on. Your monitoring/performance testing tool needs allow you to track a request through the system even as it bounces between different services.
- **Be rigorous in handling failures** (consider using Hystrix, for example, to bake in better resiliency).
- **Be flexible at staffing** and organizational design for microservices. Consider whether there are enough people for one team per service.

There is increasing interest in microservices and containers, and for good reasons. However, businesses need to make sure they have the skills and knowledge for overcoming the challenges of managing these technologies reliably, at scale. It is critical to plan and model your software delivery strategy, and align its objectives with the right skill sets and tools, so you can achieve the faster releases and reduced overhead that microservices and containers can offer.

ANDERS WALLGREN is Chief Technology Officer at Electric Cloud. Anders has over 25 years' experience designing and building commercial software. Prior to joining Electric Cloud, he held executive positions at Aceva, Archistra, and Impresse and management positions at Macromedia (MACR), Common Ground Software, and Verity (VRTY), where he played critical technical leadership roles in delivering award-winning technologies such as Macromedia's Director 7. Anders holds a B.SC from MIT.

Diving Deeper

INTO DEVOPS

TOP #DEVOPS TWITTER FEEDS

To follow right away

 @martinfowler

 @auxesis

 @bridgetkromhout

 @RealGeneKim

 @ChrisShort

 @nicolefv

 @UberGeekGirl

 @damonedwards

 @kelseyhightower

 @botchagalupe

TOP DEVOPS REFCARDZ

Getting Started With Git

dzone.com/refcardz/getting-started-git

This updated Refcard explains why so many developers are migrating to this exciting platform. Learn about creating a new Git repository, cloning existing projects, the remote workflow, and more to pave the way for limitless content version control.

Continuous Delivery Patterns

dzone.com/refcardz/continuous-delivery-patterns

Functional programming is a software paradigm that will radically change the way in which you approach any programming endeavor. Combining simple functions to create more meaningful programs is the central theme of functional programming.

Getting Started With Docker

dzone.com/refcardz/getting-started-with-docker-1

Teaches you typical Docker workflows, building images, creating Dockerfiles, and includes helpful commands to easily automate infrastructure and contain your distributed application.

DEVOPS PODCASTS

DevOps Café

Arrested DevOps

The Food Fight Show

DEVOPS ZONES

Learn more & engage your peers in our Security-related topic portals

DevOps

dzone.com/devops

DevOps is a cultural movement supported by exciting new tools that is aimed at encouraging close cooperation within cross-disciplinary teams of developers, and IT operations, and system admins.
The DevOps Zone is your hot spot for news and resources about Continuous Delivery, Puppet, Chef, Jenkins, and more.

Cloud

dzone.com/cloud

The Cloud Zone covers the host of providers and utilities that make cloud computing possible and push the limits (and savings) with which we can deploy, store, and host applications in a flexible, elastic manner. The Cloud Zone focuses on PaaS, infrastructures, security, scalability, and hosting servers.

Agile

dzone.com/agile

In the software development world, Agile methodology has overthrown older styles of workflow in almost every sector. Although there are a wide variety of interpretations and techniques, the core principles of the Agile Manifesto can help any organization in any industry improve their productivity and success. The Agile Zone is your essential hub for Scrum, XP, Kanban, Lean Startup, and more.

DEVOPS BOOKS

The DevOps Handbook

by Gene Kim, Jez Humble, Patrick Debois, and John WIllis

DevOps for Developers

by Michael Hüttermann

Release It!: Design and Deploy Production-Ready Software

by Michael T. Nygard


Continuous
Delivery
Develop. Test. Release.
Deliver high-quality software, faster, with
a fully automated software pipeline.
Only CA provides an integrated and open
continuous delivery ecosystem.
Explore ca.com/continuous-delivery
ca
technologies

It's amazing to think of the change in economic behavior over past 20 years. In 1995—the year both Amazon and eBay launched—virtually all commerce was conducted in the physical realm.

Today that's changed. In the application economy, customers' impressions are overwhelmingly shaped by their interactions with your web and mobile applications. The battleground for consumer loyalty is no longer in the physical world: it takes place on your web and mobile apps. This means that whatever products or services your company sells—and whether you realize it or not—your company is in the software business.

To compete in the application economy, your organization has to create software the way a modern factory manufactures goods. Specifically, software needs to be developed faster, at lower costs, and with high degree of quality. And this is true across all virtual touchpoints: your public-facing web and mobile apps, as well as your backend systems, are all equally critical to delivering a superior customer experience.

Agile development methodologies are a step in the right direction. DevOps takes things one step further. But the ultimate goal of any organization should be transforming into a factory capable of continuously delivering software.

Continuous delivery is not an easy task. It requires automation throughout the software development lifecycle, as a bottleneck anywhere can back up the entire assembly line. That means development, testing and release automation all must occur continuously—and concurrently. Testing is often the last hurdle to continuous delivery, and achieving continuous testing means shift-left testing practices, test automation, and testing at the API level.

CA offers an open and integrated portfolio of continuous delivery solutions that automate software delivery—from planning through production. These solutions help you accelerate the delivery of innovative, high-quality applications to drive competitive advantage and win in the application economy.

WRITTEN BY BRENDAN HAYES
DIRECTOR OF DEVOPS SOLUTIONS MARKETING, **CA TECHNOLOGIES**

PARTNER SPOTLIGHT

Continuous Delivery solutions By CA Technologies

ca technologies

Where development, testing, and release teams can work in a unique and open integrated ecosystem with proven results.

CATEGORY
DevOps and Continuous Delivery

NEW RELEASES
Continuous

OPEN SOURCE
Yes

CASE STUDY

GM Financial, the finance arm of General Motors, was striving to deliver applications and updates faster, to better serve its customers—and make it easier for them to get loans. Rapid growth within in the business and increased competition meant traditional development methods were no longer keeping pace.

GM Financial recognized the need to deliver higher quality software, faster—something that could only be achieved by automating much of the software development lifecycle.

A cohesive effort spanning dev, ops, and quality assurance did just this. With the help of CA's continuous delivery solutions, GM Financial was able to shorten a standard server deployment from several hours to a few minutes. Results like this have had a direct customer-facing impact: now loans can be processed in 1 day vs. 1-2 weeks.

Listen to their story here.

STRENGTHS

- Integrated and open, end-to-end continuous delivery ecosystem
- Develop continuously to release applications up to 20x faster
- Test continuously to gain up to a 25% reduction in testing cost and time

NOTABLE CUSTOMERS

- SunTrust
- Direct Line Group
- RaboBank
- Manheim
- AutoTrader
- Jewelers Mutual

BLOG blogs.ca.com

TWITTER @CAInc

WEBSITE ca.com/continuous-delivery

What the Military Taught Me About DevOps

BY **CHRIS SHORT**

GLOBAL DEVOPS ENGINEER AT **SOLARWINDS MSP**

QUICK VIEW

01 Failing fast is not unique to DevOps but it is an incredibly important strategy to adhere to in DevOps

02 The US military has numerous lessons learned from some of its failures that can be brought into the DevOps space

03 One nuclear mishap in 1961 demonstrates issues DevOps engineers face today

04 Practicing failures and mitigating issues as a result of those failures should be a standard function of DevOps teams

When we talk about failures in DevOps we are usually talking about how we want to fail fast. What is failing fast though? The term "fail fast" has several origins but the one that is referred to in Agile is probably the most appropriate for DevOps. Fail fast is a strategy in which you try something, it fails, feedback is delivered quickly, you adapt accordingly, and try again. Failing fast is the only way to fail in DevOps. Roll out a product, service, or application quickly, and if it doesn't pan out, move on quickly.

We don't often think of military failures in a positive light, and rightfully so. The Battle of the Little Bighorn did not work out well for General Custer. Operation Eagle Claw, the aborted operation to rescue hostages in Iran, resulted in loss of life and contributed to President Jimmy Carter's failed re-election bid. And a nuclear mishap almost resulted in the creation of "a very large Bay of North Carolina," according to Dr. Jack ReVelle, Explosive Ordinance Disposal officer. For the scope of this article, we'll focus on this near-nuclear B-52 blunder, exploring what went wrong and applying the lessons learned from it to our DevOps craft today.

The 1961 Goldsboro B-52 crash is what I consider a near miss for humanity and a very well timed wake-up call for the US military, who nearly bombed its own country. Around midnight on January 24, 1961 a B-52G Stratofortress was flying a Cold War alert flight out of Seymour-Johnson Air Force Base, North Carolina. These alert flights were part of the US military's answer to what was believed to be a superior Soviet ballistic missile threat. The B-52G that took off that night in '61 had two Mark 39 thermonuclear weapons on board.

The Cold War was a very tenuous time in world history. The battle over who could deploy weapons fastest between east and west was far greater than any Vim vs. Emacs flamewar could ever hope to reach; it was almost unquantifiable. The US military commanders in charge of the nuclear weapons were so afraid of not being able to respond to an attack, that they routinely fought against the use of safeties in nuclear weapons. We can draw a parallel here to eager investors wanting to see return on investment or managers trying to meet unrealistic delivery dates.

Continuing our analogy, the role of development and operations will be played by the scientists and bomb makers of the era, who wanted weapons to fail safely (thus not exploding when something went wrong). The makers wanted safe backout plans laid out as part of the design, planning, and implementation. Meanwhile, the investors/managers (military commanders) wanted weapons that could be deployed quickly and cheaply. The makers and commanders were at odds in their philosophies. As a result, the Mark 39 bomb had safeties disabled when the aircraft carrying them was aloft. On

this night in 1961, the B-52G carrying these weapons above Faro, North Carolina had a structural failure and broke up in mid-air.

The two Mark 39 bombs, clocking in at 3.8 megatons a piece (more than 250 times the destructive power of the Hiroshima bomb), plummeted to earth. One Mark 39 deployed its parachute (a part of a planned detonation of the weapon) and it was later discovered that three of the four safety mechanisms were flipped off during the accident; one step away from a nuclear explosion. This particular bomb landed in a tree and was safely recovered.

The other Mark 39 bomb did not deploy its parachute. Instead, it became a nuclear lawn dart and slammed in to a swampy patch of earth at an estimated 700 miles per hour breaking apart on impact. The core (or pit) of the bomb was safely recovered. A complete recovery of all of the weapon's components was not possible due to the conditions. As a result, a small chunk of eastern North Carolina has some fissile material leftover from the accident. There is a concrete pad in place to prevent tampering and a hastily written law that no farming or other activity will take place deeper than five feet at the site. There was no disaster recovery plan, and it showed.

The 1961 Goldsboro B-52 Crash is a prime example of failing fast going wrong. We were one step away from not one but two multi-megaton nuclear detonations on the US eastern seaboard. The US military did not want inert bombs to fall on the Soviet Union if a delivery system was somehow disabled. The investors/managers wanted the weapons to fail spectacularly. The development and operations teams wanted the weapons to fail fast and safely. Investors/managers did not want to bother with security testing, failure scenarios, and other DevOps-type planning, and this nearly resulted in catastrophic costs.

Luckily, for most of us following DevOps practices, we do not have lives or the fate of humanity in our hands. We can deploy things like Chaos Monkey in our production environments with little risk to life and limb. If you break something in stage, is it not doing its intended purpose: to catch bugs safely before they manifest themselves in production? Take advantage of your dev, test, and stage environments. If those non-production environments are not easily rebuilt, do the work to make them immutable. Automate their deployment so you can take the time to rigorously test your services. Practice failures vigorously; spend the time needed to correct or automate issues out of the systems.

Following the near disaster in Goldsboro, the US government conducted an amazingly detailed postmortem. Speaking with Dr. Jack ReVelle, the Explosive Ordnance Disposal (EOD) officer who responded to the accident, I learned that significant improvements were made to training and documentation. Security and safety became an iterative part of the development and deployment processes. Prior to this deployment, Dr. ReVelle was not trained in disaster recovery of nuclear devices, "We were writing the book on it as we went." As a result of this accident, techs were taught how to manage the systems before deployment. Additionally, documentation was updated continuously to include necessary information about the systems as they were being developed. Better tooling was procured for the teams to manage incidents. In short, significant rigor was added to training programs, deployment plans, and documentation processes in the EOD teams. EOD teams now planned for failures in the way DevOps teams of today plan for failures.

> "The most important part of failing is not the fact something failed, but how you respond to and learn from such failures."

One thing you have to keep in mind about everything procured by the US government is that it is delivered by the person or company that is the lowest bidder. In other words, a failure rate is expected with anything. The military has to practice failing in any and all forms that can be thought of just like DevOps professionals should. Practicing failures is important because it will improve your recognition time and response times to these failures. Your teams will build a sort of "muscle memory" to effectively quash issues before they become incidents. This "muscle memory" will allow your teams to iterate through one scenario while discussing other scenarios more calmly. Common sense is not common, so explicit documentation and processes are incredibly important. Remember, the most important part of failing is not the fact something failed, but how you respond to and learn from such failures.

CHRIS SHORT has over two decades in various IT disciplines from textile manufacturing to dial-up ISPs to DevOps Engineer. He's been a staunch advocate for open source solutions throughout his time in the private and public sector. He's a partially disabled US Air Force Veteran living with his wife and son in NC. Check out his writing at chrisshort.net.

Bridge the gap between Dev and Ops with Jobs-as-Code

Control-M & DevOps: Automation API

—

Bring IT to Life at bmc.com/jobsascode

End the Hidden Drag in Application Delivery

DevOps is about increasing the speed of delivering high-quality applications. However, many DevOps teams do not realize how much time they lose dealing with the mix-and-match job scripts that are rolling through to production.

During the build phase, simple tools and manual scripts are used to code jobs for applications. This non-standard approach results in downstream issues when Operations goes to scale in production. Operations teams are forced to send back to Dev or rework the scripts so applications can run as expected. Often this means converting scripts to comply with the enterprise job scheduling solution. This extra effort causes unnecessary delays in the application delivery process.

With agile development and delivery top priority for companies worldwide, job scheduling and file transfer solutions are being extended to support these new initiatives. To avoid rework, headaches, and poor quality applications, BMC is working with Dev teams around the globe to shift-left these functions from Ops into Dev with Jobs-as-Code. Control-M Jobs-as-Code gives developers access to the power of Control-M by embedding job scripts into the code. The functionality expected in production is now delivered from development through the entire lifecycle.

> 70% of business processing is done by application job scheduling. Jobs-as-Code increases quality and shortens delivery time.

70% of business processing is done by application job scheduling. Jobs-as-Code increases quality and shortens delivery time (source: How to Accelerate DevOps Delivery Cycles, 2016). Free up development time to rapidly improve applications running in production. The result is a faster, better DevOps delivery cycle.

WRITTEN BY TINA STURGIS
DIRECTOR, SOLUTION MARKETING AT **BMC**

PARTNER SPOTLIGHT

Control-M By BMC

"When we look at how much work we push through our environment running multiple platforms, we estimate that we would need 30 full-time personnel working seven days a week to do the amount of work that our team of six does with Control-M." — **DON SNIOS**, SENIOR MANAGER OF OPERATIONS AND SCHEDULING, INGRAM MICRO

CATEGORY
Enterprise Job Scheduling / Jobs-as-Code

NEW RELEASES
Continuous

OPEN SOURCE
No

CASE STUDY

Unum, which encompasses Unum US, Colonial Life, Starmount, and Unum UK, is a leading provider of financial protection benefits in the workplace, including disability, life, accident, and critical illness coverage. The IT organization at Unum is committed to putting the right technologies in place to support the company's growth. BMC Control-M has been part of that effort for more than 20 years. Today, Control-M manages every aspect of their batch processing—supporting internal business operations, enrolling policyholders, transferring employee data reliably among employers, Unum, and payroll providers, and processing claims for individuals. The result has been a **60 percent reduction in batch service requests coming into the scheduling team**. They've also seen a **900 percent increase in jobs submitted via Self Service over the past 3½ years**.

STRENGTHS

- **Jobs-as-Code** helps organizations operationalize applications faster by embedding automated job scheduling into the current development and release process.
- **Strengthen application quality** by decreasing downstream complexity and minimizing errors when rolling into production.
- **Increase team member productivity** by reducing the time spent learning a new scheduling tool for every application or manually scripting and re-scripting jobs.
- **Accelerate application change and deployment** cycle times with automated application workflow between test and production.
- **Quickly and reliably deliver business services** by easily connecting applications and workflow processes.

NOTABLE CUSTOMERS

- Eaton
- Carfax
- UNUM
- Ingram Micro
- British Sky Broadcasting
- Raymond James Financial

BLOG bit.ly/2h64GBL

TWITTER @ControlM_BMC

WEBSITE bmc.com/jobsascode

Solutions Directory

This directory contains software configuration, build, container, repository, monitoring, and application performance management tools, as well as many other tools to help you on your journey toward Continuous Delivery. It provides the company name, product information, open source data, and product category information gathered from vendor websites and project pages. Solutions are selected for inclusion based on several impartial criteria, including solution maturity, technical innovativeness, relevance, and data availability.

COMPANY NAME	PRODUCT	CATEGORY	OPEN SOURCE	WEBSITE
Amazon	Amazon ECS	Container Management	No	aws.amazon.com/ecs
Apache Software Foundation	Apache Ant	Build Management	Yes	ant.apache.org
Apache Software Foundation	Apache Archiva	Repository Management	Yes	archiva.apache.org
Apache Software Foundation	Apache Maven	Build Management	Yes	maven.apache.org
Apache Software Foundation	Apache Subversion	Software Configuration Management	Yes	subversion.apache.org
Apache Software Foundation	JMeter	Web and Java Testing	Yes	jmeter.apache.org
AppDynamics	AppDynamics	Application Performance Management	No	appdynamics.com
Appium	Appium	Automated Web and Mobile Testing	Yes	appium.io
Atlassian	Bamboo	Continuous Integration, Application Release Automation	No	atlassian.com/software/bamboo
Attunity	RepliWeb for ARA	Application Release Automation	No	attunity.com/products/repliweb
Automic	Automic V12	Application Release Automation	Yes	automic.com/products/application-release-automation
BigPanda	Altert Correlation Platform	Monitoring Alert Software	No	bigpanda.io
BMC	Control-M	Enterprise Job Scheduling	No	bmc.com/it-solutions/control-m.html
Buildbot	Buildbot	Continuous Integration	Yes	buildbot.net
CA	CA Release Automation	Application Release Automation	Yes	ca.com/us/products/ca-release-automation.html
Chef	Chef Automate	Continuous Deployment Platform	Yes	chef.io/automate

COMPANY NAME	PRODUCT	CATEGORY	OPEN SOURCE	WEBSITE
Chef	Continuous Delivery	Application and Infrastructure Release Automation	No	chef.io/solutions/continuous-delivery
CircleCI	CircleCI	Continuous Integration	No	circleci.com
Cloudbees	Jenkins Enterprise	Continuous Integration, Application Release Automation	No	go.cloudbees.com/docs/cloudbees-documentation/cje-install-guide
Codeship	ParallelCI	Continuous Integration	No	codeship.com/features/parallelci
Cucumber	Cucumber	Automated Rails Testing	Yes	cucumber.io
Datadog	Datadog	IT Stack Performance Management	No	datadoghq.com
Docker	Docker Swarm	Container Management	Yes	docker.com/products/docker-swarm
Eclipse	Hudson	Continuous Integration	Yes	eclipse.org/hudson
ElasticBox	ElasticBox	Container Management	No	elasticbox.com
Electric Cloud	ElectricFlow	Application Release Automation	No	electric-cloud.com/products/electricflow
FitNesse	FitNesse	Acceptance Testing Framework	Yes	fitnesse.org
Free Software Foundation	Concurrent Versions Systems (CVS)	Software Configuration Management	Yes	savannah.nongnu.org/projects/cvs
Git	Git	Software Configuration Management	Yes	git-scm.com
Gitlab	GitLab	Code Review, Continuous Integration, Continuous Delivery	Yes	about.gitlab.com
Gradle	Gradle	Build Automation	Yes	gradle.org
Grid Dynamics	Agile Software Factory	Application Release Automation	No	griddynamics.com/blueprints
Gridlastic	Gridlastic	Automated Web Testing	No	gridlastic.com
Hashicorp	Vagrant	Configuration Management	Yes	vagrantup.com
HPE	ALM Octane	Application Lifecycle Management	No	saas.hpe.com/en-us/software/alm-octane
IBM	Rational	Software Configuration Management	No	01.ibm.com/software/rational/strategy
IBM	Urbancode Build	Continuous Integration, Build Management	No	developer.ibm.com/urbancode/products/urbancode-build/
IBM	Urbancode Deploy	Application Release Automation	No	developer.ibm.com/urbancode/products/urbancode-deploy
Inedo	Buildmaster	Application Release Automation	No	inedo.com/buildmaster

COMPANY NAME	PRODUCT	CATEGORY	OPEN SOURCE	WEBSITE
Inflectra	Rapise	Automated Web Testing	No	inflectra.com/Rapise
Jenkins	Jenkins	Continuous Integration	Yes	jenkins-ci.org
JetBrains	TeamCity	Continuous Integration, Application Release Automation	No	jetbrains.com/teamcity
jFrog	Artifactory	Repository Management	No	jfrog.com/artifactory
Junit	JUnit	Unit Testing Framework	Yes	junit.org
Librato	Librato	Monitoring Alert Software	No	librato.com
Mercuiral	Mercurial	Software Configuration Management	Yes	mercurial-scm.org
Micro Focus	AccuRev	Software Configuration Management	No	microfocus.com/products/change-management/accurev
Microsoft	Team Foundation Server	Software Configuration Management	No	visualstudio.com/en-us/news/releasenotes/tfs2017-relnotes
MidVision	RapidDeploy	Application Release Automation	No	midvision.com/product/rapiddeploy
Nagios	Nagios Core	Infrastructure Monitoring	Yes	nagios.org/projects/nagios-core
New Relic	New Relic	Application Performance Management	No	newrelic.com
NuGet	NuGet	Repository Management	Yes	nuget.org
Nunit	NUnit	Unit Testing Framework	Yes	nunit.org
OpsGenie	OpsGenie	Monitoring Alert Software	No	opsgenie.com
PagerDuty	PagerDuty	Monitoring Alert Software	No	pagerduty.com
Parasoft	Parasoft	Automated Web and API Testing	No	parasoft.com
Perforce	Helix	Software Configuration Management	No	perforce.com/helix
Plutora	Plutora	Application Release Automation	No	plutora.com
Puppet Labs	Puppet	Configuration Management	No	puppetlabs.com
Rake	Rake	Build Automation	Yes	github.com/ruby/rake
Ranorex	Ranorex	Automated Web and Desktop Testing	No	ranorex.com
Red Gate Software	DLM Automation Suite	Database CI and Release Automation	No	red-gate.com/products/dlm/dlm-automation

COMPANY NAME	PRODUCT	CATEGORY	OPEN SOURCE	WEBSITE
Red Hat	Ansible	Configuration Management, Application Release Automation	Yes	ansible.com
Rocket Software	ALM and DevOps	Application Release Automation	No	rocketsoftware.com/product-categories/application-lifecycle-management-and-devops?rcid=mm-pr-alm-explore
Sahi	Sahi	Automated Web Testing	No	sahipro.com
SaltStack	Salt	Configuration Management	Yes	saltstack.com
Sauce Labs	Sauce Labs	Automated Web and Mobile Testing	Yes	saucelabs.com
Selenium	Selenium WebDriver	Automated Web Testing	Yes	seleniumhq.org
Serena	Serena Deployment Automation	Application Release Automation	No	serena.com/index.php/en/products/deployment-configuration-automation/serena-deployment-automation/overview
SmartBear Software	SoapUI	Automated Web and API Testing	Yes	soapui.org
SnapCI	SnapCI	Continuous Integration	No	snap-ci.com
Solano Labs	Solano Labs	Continuous Integration	No	solanolabs.com
Sonatype	Nexus	Repository Management	Yes	sonatype.org/nexus
Tellurium	Tellurium	Automated Web Testing	No	te52.com
TestingBot	TestingBot	Automated Web Testing	No	testingbot.com
TestNG	TestNG	Unit Testing Framework	Yes	testng.org
The Linux Foundation	Kubernetes	Container Management	Yes	kubernetes.io
Thoughtworks	Go	Application Release Automation	Yes	gocd.io
TravisCI	TravisCI	Continuous Integration	Yes	travis-ci.org
VictorOps	VictorOps	Monitoring Alert Software	No	victorops.com
Watir	Watir	Automated Web Testing	Yes	watir.com
WeaveWorks	Weave Cloud	Containers, Microservices, and SaaS	No	weave.works/solution/cloud
Windmill	Windmill	Automated Web Testing	Yes	getwindmill.com
Xebia Labs	XL Deploy	Application Release Automation	No	xebialabs.com/products/xl-deploy
xUnit	xUnit	Unit Testing Framework	Yes	xunit.github.io
Zend	Zend Server	Application Release Automation for PHP	No	zend.com/en/products/zend_server

GLOSSARY

ARTIFACT Any description of a process used to create a piece of software that can be referred to, including diagrams, user requirements, and UML models.

AUTONOMY The ability to make changes with the resources currently available, without the need to defer to something or someone higher up in the hierarchy.

BRANCHING The duplication of an object under review in source control so that the same code can be modified by more than one developer in parallel.

COMMIT A way to record the changes to a repository and add a log message to describe the changes that were made.

COMPLEX-ADAPTIVE SYSTEMS Any system made of a collection of similar, smaller pieces that are dynamically connected and can change to adapt to changes for the benefit of a macro-structure.

CONTAINERS Resource isolation at the OS (rather than machine) level, usually (in UNIX-based systems) in user space. Isolated elements vary by containerization strategy and often include file system, disk quota, CPU and memory, I/O rate, root privileges, and network access. Much lighter-weight than machine-level virtualization and sufficient for many isolation requirement sets.

CONTINUOUS DELIVERY A software engineering approach in which continuous integration, automated testing, and automated deployment capabilities allow software to be developed and deployed rapidly, reliably, and repeatedly with minimal human intervention.

CONTINUOUS DEPLOYMENT A software development practice in which every code change goes through the entire pipeline and is put into production automatically, resulting in many production deployments every day. It does everything that Continuous Delivery does, but the process is fully automated, and there's no human intervention at all.

CONTINUOUS INTEGRATION A software development process where a branch of source code is rebuilt every time code is committed to the source control system. The process is often extended to include deployment, installation, and testing of applications in production environments.

CONTINUOUS QUALITY A principle that preaches the continuous quest for quality across the entire SDLC, starting from requirements definition, code development, testing, and operations. Another key area of focus for Continuous Quality is the application code pipeline orchestration. There are many opportunities to negatively impact the quality of an application when code is being manually moved across environments.

CONTINUOUS TESTING The process of executing unattended automated tests as part of the software delivery pipeline across all environments to obtain immediate feedback on the quality of a code build.

DEPLOYMENT PIPELINE A deployment pipeline is an automated manifestation of your process for getting software from version control into the hands of your users. (source: informIT.com)

DEVOPS An IT organizational methodology where all teams in the organization, especially development teams and operations teams, collaborate on both development and deployment of software to increase software production agility and achieve business goals.

EVENT-DRIVEN ARCHITECTURE A software architecture pattern where events or messages are produced by the system, and the system is built to react, consume, and detect other events.

FAIL FAST A strategy in which you try something, it fails, feedback is delivered quickly, you adapt accordingly, and try again.

MICROSERVICES ARCHITECTURE The practice of developing software as an interconnected system of several independent, modular services that communicate with each other.

MODEL-BASED TESTING A software testing technique in which the test cases are derived from a model that describes the functional aspects of the System Under Test (SUT). Visual models can be used to represent the desired behavior of a SUT, or to represent testing strategies and a test environment. From that model manual tests, test data, and automated tests can be generated automatically.

ONE-STOP SHOP / OUT-OF-THE-BOX TOOLS Tools that provide a set of functionalities that works immediately after installation with hardly any configuration or modification needs. When applied to the software delivery, a one-stop shop solution allows quick setup of a deployment pipeline.

PAIR PROGRAMMING A software development practice where two developers work on a feature, rather than one, so that both developers can review each others' code as it's being written in order to improve code quality.

PRODUCTION The final stage in a deployment pipeline where the software will be used by the intended audience.

SOURCE CONTROL A system for storing, tracking, and managing changes to software. This is commonly done through a process of creating branches (copies for safely creating new features) off of the stable master version of the software and then merging stable feature branches back into the master version. This is also known as version control or revision control.

STAGING ENVIRONMENT Used to test the newer version of your software before it's moved to live production. Staging is meant to replicate as much of your live production environment as possible, giving you the best chance to catch any bugs before you release your software.

TECHNICAL DEBT A concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution.

TEST AUTOMATION The use of special software (separate from the software being tested) to control the execution of tests and the comparison of actual outcomes with predicted outcomes.

INTRODUCING THE

Security Zone

Arm Yourselves From Hacks and Data Leaks!

Retroactively injecting security into applications is unnecessary and expensive. To avoid these costly patches, secure development has become essential for preventing hacks.

Our newest Zone addresses security issues head-on, offering tools, tutorials, and updates to secure your applications and the machines that run them!

NETWORK SECURITY

WEB APPLICATION SECURITY

DENIAL OF SERVICE ATTACKS

IoT SECURITY

dzone.com/security