THE DZONE GUIDE TO

CONTINUOUS DELIVERY

VOLUME III

BROUGHT TO YOU IN PARTNERSHIP WITH

CloudBees
The Enterprise Jenkins Company

Gradle

JET BRAINS

JFrog

Microsoft

redgate
ingeniously simple

shippable

ThoughtWorks®

DEAR READER,

To paraphrase Kent Beck: software delivers no value apart from runtime. Ideas take physical form in hardware, virtualized only part of the way down, and someone other than the developers makes the ideas manifest. So then: ops folks are the crop's caretakers; developers design the seeds.

Well, of course this isn't true. Developers don't turn pie-in-the-sky math into socially constructive math that sysadmins then make useful. No: developers write code that makes things happen. We care about runtime just as much as ops, so we need to know how our software is helping people in reality—and how it can do a better job—in order for us to refine both the ideas and the implementation.

So cycle-time shortening—the metric of continuous delivery, and one of the Twelve Principles of Agile Software—is equally about users and makers. Working software delivered sooner is better for users than equally working software delivered later. And informative feedback gleaned more quickly from real-world use is better for developers than equally informative feedback gathered more slowly. Your customers' problem space measures your software's value better than your requirements specification ever could. Calibrate against that space as often as you can.

But not all defects appear immediately (sometimes not even for years), and the real-world harm caused by certain defects far outweighs the benefits of quicker delivery. So doing DevOps right involves tradeoffs, risk management, and decisions about the future, all made in uncertainty. And real-world experience in Continuous Delivery matters a lot.

To help deliver this hard-won experience to our readers, we're publishing our latest Guide to Continuous Delivery, full of original research and expert advice on modern challenges (iterative roadmapping, security) and solutions (containers, microservices) for DevOps professionals striving for the CD ideal.

Read, design, implement, refine, repeat—and let us know what you think.

JOHN ESPOSITO
EDITOR-IN-CHIEF
RESEARCH@DZONE.COM

TABLE OF CONTENTS

3 **EXECUTIVE SUMMARY**

4 **KEY RESEARCH FINDINGS**

8 **ADDING ARCHITECTURAL QUALITY METRICS TO YOUR CD PIPELINE** BY ANDREW PHILLIPS

12 **HOW TO DEFINE YOUR DEVOPS ROADMAP** BY MIRCO HERING

15 **A SCORECARD FOR MEASURING ROI OF A CONTINUOUS DELIVERY INVESTMENT**

16 **WELCOME TO PAIN-SYLVANIA** INFOGRAPHIC

18 **CONTINUOUS DELIVERY & RELEASE AUTOMATION FOR MICROSERVICES** BY ANDERS WALLGREN

21 **DIVING DEEPER INTO CONTINUOUS DELIVERY**

22 **THE CONTINUOUS DELIVERY MATURITY CHECKLIST**

24 **CONTINUOUS DELIVERY FOR CONTAINERIZED APPLICATIONS** BY JIM BUGWADIA

30 **SECURING A CONTINUOUS DELIVERY PIPELINE** BY MATTHEW SKELTON & MANUEL PAIS

32 **EXECUTIVE INSIGHTS ON CONTINUOUS DELIVERY** BY TOM SMITH

35 **CONTINUOUS DELIVERY SOLUTIONS DIRECTORY**

39 **GLOSSARY**

EDITORIAL

Andre Powell
PROJECT COORDINATOR

John Esposito
research@dzone.com
EDITOR-IN-CHIEF

G. Ryan Spain
DIRECTOR OF PUBLICATIONS

Mitch Pronschinske
SR. RESEARCH ANALYST

Matt Werner
MARKET RESEARCHER

Moe Long
MARKET RESEARCHER

Michael Tharrington
EDITOR

Tom Smith
RESEARCH ANALYST

BUSINESS

Rick Ross
CEO

Matt Schmidt
PRESIDENT & CTO

Jesse Davis
EVP & COO

Kellet Atkinson
VP OF MARKETING

Matt O'Brian
DIRECTOR OF BUSINESS DEVELOPMENT

Alex Crafts
sales@dzone.com
DIRECTOR OF MAJOR ACCOUNTS

Chelsea Bosworth
MARKETING ASSOCIATE

Caitlin Zucal
MARKETING ASSOCIATE

Aaron Love
MARKETING ASSOCIATE

Chris Smith
PRODUCTION ADVISOR

Jim Howard
SR ACCOUNT EXECUTIVE

Chris Brumfield
ACCOUNT MANAGER

ART

Ashley Slate
DESIGN DIRECTOR

SPECIAL THANKS to our topic experts Andrew Phillips, Mirco Hering, Anders Wallgren, Jim Bugwadia, Matthew Skelton, Manuel Pais and our trusted DZone Most Valuable Bloggers for all their help and feedback in making this report a great success.

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.

EXECUTIVE SUMMARY

Ever since Flickr educated the world on how to achieve ten deployments a day, and Jez Humble and Dave Farley wrote *Continuous Delivery*, the technology industry has been making more and more strides towards implementing DevOps methodologies, deploying code even thousands of times a day. This practice increases collaboration between development and operations teams in order to deploy code to users as soon as it's ready, shortening the feedback loop between customers and businesses and allowing organizations to adapt to constantly changing consumer needs. This is not just a practice for developing software, but a practice for how the whole organization should run to put users first. The *DZone 2016 Guide to Continuous Delivery* provides insight into how DevOps has been affected by new technological advancements in recent years, the obstacles facing developers, and the status of DevOps in the Enterprise. **The resources in this guide include:**

- Expert knowledge for implementing Continuous Delivery and DevOps best practices.
- A directory of tools to consider when implementing a Continuous Delivery environment.
- A checklist for calculating the maturity of Continuous Delivery pipelines and culture, as well as the real value of implementing DevOps practices.
- Analysis of trends in the space based on feedback from almost 600 IT professionals.

KNOWLEDGE ABOUT CONTINUOUS DELIVERY HAS INCREASED

DATA

Our 2015 Guide to Continuous Delivery reported that 50% of professionals believed their organizations had not achieved Continuous Delivery, down 9% from 2013. This year, that percentage rose 9% from last year. Furthermore, 30% this year said they implemented Continuous Delivery for some projects, while 11% said they had implemented Continuous Delivery across the whole organization. This shows a 6% and 3% decrease from last year, respectively. Once these positive responses were filtered by criteria for achieving Continuous Delivery (software in a shippable state before deployment, whether a team performed push-button or fully automated deployments, and if all stakeholders had visibility into the software delivery environment), the number of professionals who believed their organizations had achieved Continuous Delivery for some or all projects dropped to 10%.

IMPLICATIONS

However, 36.6% of all organizations use both Dev and Ops teams to deploy code, as opposed to 20.6% last year. This seems to indicate that more people have learned more about the tenets of Continuous Delivery as time has passed, and while improvements have been made, there is an understanding that there is still a long way to go.

RECOMMENDATIONS

To achieve Continuous Delivery, organizations will need to ensure that all key stakeholders are involved in its success, and that means solidifying a DevOps adoption strategy. For further reading on defining and following through on a DevOps roadmap, consult Mirco Hering's piece "How to Define Your DevOps Roadmap" on page 9.

DEVELOPMENT IS LEADING THE CHARGE TO CD

DATA

When it comes to adopting the tools necessary for implementing Continuous Delivery, developers are leading the charge. 91% of developers have adopted an issue tracking tool, as opposed to 68% of operations departments; 97% of developers have adopted version control, compared to 39% in operations departments; and 82% of developers use CI tools—such as Jenkins or Bamboo—while only 28% of operations departments have done the same. There is a little more parity in adoption among configuration management tools, where 37% of operations departments and 32% of developers use tools like Chef or Puppet.

IMPLICATIONS

Overall, it seems that developers are much more likely to adopt tools that facilitate Continuous Delivery. Whether developers are more flexible and willing to try new tools or are more in-touch with current technology is difficult to say, but it is clear that operations teams are not keeping up with developers, which is automatically detrimental to DevOps practices, since the teams are not united in their tooling and processes. Collaboration on deployments between developers and operations has increased since last year's survey, but not when it comes to the toolchain.

RECOMMENDATIONS

One way to bridge this gap may be through container technology, which has been adopted by 17% of respondents, and affects how applications are tested and developed as well as how they're deployed in some organizations. Jim Bugwadia has contributed an article on Continuous Delivery for Containerized Applications on page 24 in this research guide.

DEVOPS SKILLS AND CULTURE NEED TO BE NURTURED

DATA

The lack of a supportive company culture is still a major barrier to adopting Continuous Delivery—this year 57% of our audience reported that a lack of collaboration and DevOps practices was a barrier to adopting Continuous Delivery, compared to 64% last year. 49% of this year's survey respondents identified a lack of skills as a major factor, a 3% growth from last year. A major factor in strengthening these skills is communication of accurate metrics to employees in order to measure and improve performance. However, 38% of survey respondents reported that no metrics are used to measure DevOps success.

IMPLICATIONS

As IT professionals learn more about DevOps, they can learn to pinpoint what keeps them from achieving Continuous Delivery. Still, organization-wide challenges are preventing ideal DevOps implementation, and until an entire company embraces DevOps philosophies, true Continuous Delivery cannot be achieved—developers alone, or even with Ops assistance, cannot make Continuous Delivery happen.

RECOMMENDATIONS

DevOps practices should be backed by everyone in the organization. If all stakeholders are involved in adopting Continuous Delivery, metrics and visibility into the environment can only help the organization improve. Andrew Phillips contributed a great piece on adding architectural quality metrics to your pipeline on page 6.

KEY RESEARCH FINDINGS

596 IT PROFESSIONALS RESPONDED TO DZONE'S 2016 CONTINUOUS DELIVERY SURVEY. HERE ARE THE DEMOGRAPHICS OF THIS SURVEY:

- Developers (42%) and Development Leads (25%) were the most common roles.
- 63% of respondents come from large organizations (100 or more employees) and 37% come from small organizations (under 100 employees).
- The majority of respondents are headquartered in Europe (44%) or the US (32%).
- Over half of the respondents (61%) have over 10 years of experience as IT professionals.
- We only surveyed developers who use Java in their organization. Other language ecosystems for these organizations include C# (31%), C/C++ (29%), Universal JavaScript (28%), PHP (26%), Python (26%), and Groovy (23%).

THE TEXTBOOK DEFINITION OF CONTINUOUS DELIVERY

Jez Humble and Dave Farley, the authors of the book Continuous Delivery, defined three key traits to determine when an organization has achieved Continuous Delivery. We asked our audience whether they were confident that code was in a shippable state after changes were made and any tests were completed; whether their teams perform push-button or fully automated deployments of code changes to any environment; and if all stakeholders have visibility into the software delivery environment. Of those who believe they've achieved CD for some or all projects (41%), we found that only 18% feel confident in their code's readiness, can perform deployments of code changes on-demand, and offer visibility into their entire production process. Though the sample of respondents who believed they achieved Continuous Delivery was 9% smaller from last year's survey, the 18% of respondents who reported having all three of these CD traits was the same percentage as in last year's survey. We believe the smaller number in those who believe they have implemented Continuous Delivery for some or all of their projects is a result of the increased knowledge that has been circulating as DevOps becomes a more accepted and understood term in the IT industry.

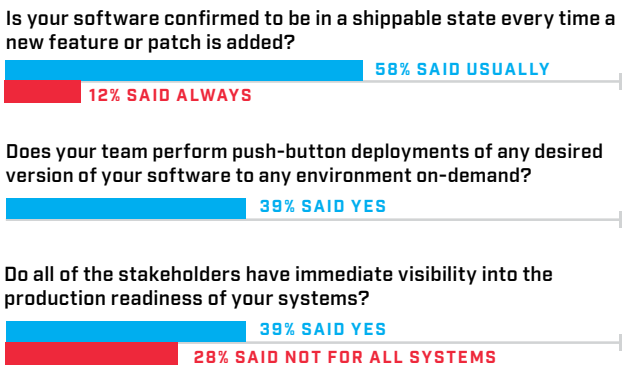
CONTAINERS ARE HOT, BUT THERE IS A LEARNING CURVE

Though container technology, like JVM containers, has existed for a long time, technologies like Docker have recently made headlines as an easier way to spin up environments by allowing the kernel of an operating system to host multiple environments. As far as overall container usage goes, 17% of respondents are currently using containers in their organization, and a majority (56%) are either considering or evaluating container technology. However, of those who are using containers, over half (53%) still have issues with configuring and setting up environments, even though this is commonly cited as a key benefit of using Docker. As the technology is still new and not yet mature, it seems there is a learning curve to using container technology.

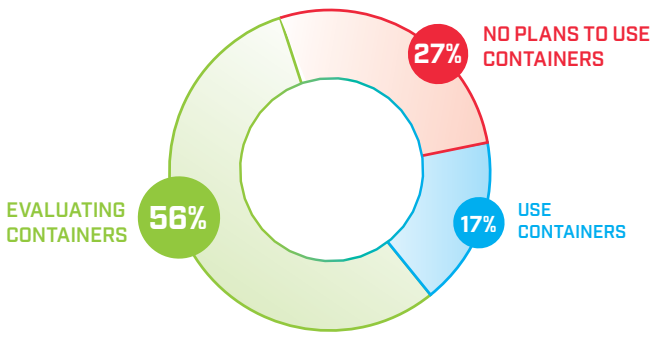
SOME COMPANIES OVERCOME BARRIERS MORE EASILY THAN OTHERS

Of the barriers to adopting Continuous Delivery, three were singled out as the most prominent: lack of a collaborative corporate culture (57%); lack of time to implement Continuous Delivery practices (51%); and lack of relevant skill sets (49%). Some respondents' answers differed largely based on the size of the organization they worked in. As shown in the graph below, on average, corporate culture was cited as a barrier by

01. **CONTINUOUS DELIVERY KEY TRAITS**



02. **CONTAINER USAGE AMONGST DEVELOPERS**



only 23% of users who worked at startups (less than 10 people). From there, a clear trend is present that indicates that more respondents recognized the lack of a supportive culture as a barrier if they worked at larger companies. On the other hand, results are more normalized for the other two barriers.

However, respondents at companies that were either very small (1-4 employees) or mid-sized (20-99 employees) were more likely to cite lack of time as a major barrier (62% and 60%, respectively), and only very small companies (1-4 employees) did not consider the lack of engineering skills to be a significant barrier (39%). We can reasonably conclude that smaller companies and startups are more likely to have issues related to time when they start implementing Continuous Delivery, if they did not begin life that way, and that as companies grow, a more silo-focused culture grows with it, making DevOps practices more difficult to adopt.

DEV AND OPS COLLABORATION IS IMPROVING

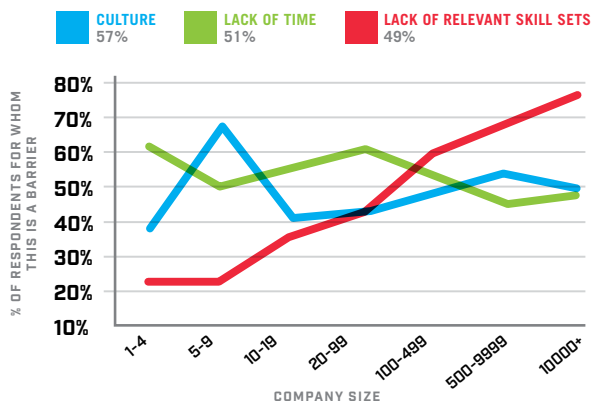
While the lack of a collaborative culture is still a significant barrier for many companies, there has been a greater effort to work between development and operations. This year, both departments deployed code together 16% more than last year's survey respondents reported (37% compared to 21%). We also discovered that specific DevOps teams could improve collaboration. Of the 32% of respondents who have a DevOps team in their organization, 57% identified that collaboration and breaking down silos was a major focus of that team. When a DevOps-focused team is in place, 41% of development and operations teams deploy code together, compared to 35% when there is no designated DevOps team. This signifies that specialized DevOps teams can be effective at driving collaboration between different departments. While company culture can still be a limiting factor when adopting Continuous Delivery, improvements are being made.

WHAT PROCESSES IMPACT COLLABORATION

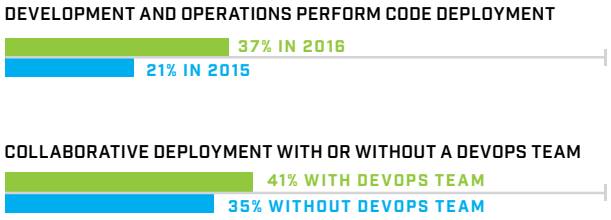
For this year's survey, we asked our users if their software delivery process included any of the following processes: builds broken up into stages, performance issue detection, security issue detection, usability issue detection, visibility to all stakeholders in the organization, a thorough audit trail, automatic checks to proceed, manual checks to proceed, automated performance testing, and automated feature validation. This was to gauge if using these techniques lead to any improvements in DevOps collaboration, and what organizations working to achieve Continuous Delivery were focused on. Of those organizations, 66% have adopted builds broken up into stages, 62% include manual checks to proceed, and 53% include automated checks to proceed. This indicates that organizations moving towards Continuous Delivery are making changes to their pipeline that will make it easier and faster to deploy changes to software in production, while maintaining high quality.

Organizations whose development and operations teams deploy code to production use these processes 15% more than organizations where only development deploys changes, and 15% more than organizations where only operations deploys changes to software. Within collaborative organizations, the two most used processes were visibility available to all stakeholders (53%) and automated feature validation (51%). This indicates that (1) automated testing and validation is gaining significant traction within teams who are trying to collaborate more; and (2) that allowing all stakeholders visibility into the process suggests that when management is more invested in creating a DevOps culture, development and operations are more likely to work together and adopt new technologies.

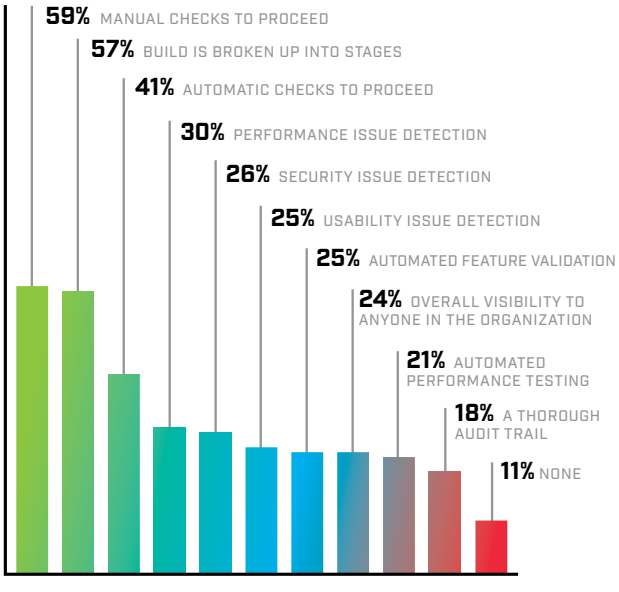
03. **CD BARRIERS AS PERCENTAGE OF COMPANY SIZE**



04. **CODE DEPLOYMENT BY DEVELOPMENT + OPERATIONS TEAMS**



05. **DOES YOUR SOFTWARE DELIVERY PROCESS INCLUDE ANY OF THE FOLLOWING?**



CloudBees®

The Enterprise Jenkins Company

Continuous Delivery Powered by Jenkins

Your DevOps Foundation

Build → Test → Stage → Deploy

The Enterprise Jenkins Company

Learn more: www.cloudbees.com/jenkins-pipeline

Five Steps to Automating Continous Delivery Pipelines with Jenkins Pipeline

When teams adopt CD, the goal is to automate processes and deliver better software more quickly. The Jenkins Pipeline plugin enables you to easily automate your entire software delivery pipeline. Following are a series of best practices to follow, to get the most out of pipeline automation.

MODEL YOUR EXISTING VALUE STREAM

You've mapped out your pipeline; now create a more detailed model to see how code flows through various steps—development, QA, preproduction and production. Identify bottlenecks and manual processes that slow things down.

MEASURE YOUR PIPELINE

Now that you've modeled your pipeline, measure everything. Measure speed. Measure the quality of builds. Test everything.

FAIL QUICKLY

If you're going to break something, break it early. If a developer commits a change, he should know within five minutes if he broke the build. Failures later in the process are more expensive, so fail early.

AVOID LONG CRITICAL PATHS

If you create a long path, it can set up a situation where jobs are waiting for things they don't need to be waiting for. Split up your work into chunks—the overall process will finish more quickly.

FIND WAYS TO RUN STAGES IN PARALLEL

One way to save time and energy is to enable jobs to run in parallel. There is no point in wasting people's time waiting on a process when machines can run tasks in the background.

DON'T OVER-AUTOMATE

Problems can arise when processes can't be manually reproduced. Pipelines need to be configured to allow for human intervention, when needed. Any benefit automation provides can quickly be undone if a failure requires a full restart of a pipeline process.

CONCLUSION

CD pipelines are like crops. You don't set them and forget them; you cultivate them so they grow and generate better harvests. These best practices will help you build better software, more quickly, returning real value to the business.

WRITTEN BY DAN JUENGST

SENIOR DIRECTOR OF PRODUCTS AT CLOUDBEES

Adding Architectural Quality Metrics to Your CD Pipeline

QUICK VIEW

01
Continuous Delivery is about shipping software better, not just more quickly.

02
As the scope/size of changes in the pipelines get smaller (and pipeline runs more frequent), validating existing behaviour becomes at least as important as verifying the new features.

03
We need to go beyond automated *functional* testing and add stress and performance testing into our pipeline to validate the behaviour of an application at scale.

BY ANDREW PHILLIPS

"At the end of each sprint." "4 times a day." "Every 6 minutes." Pick a handful of blog posts or presentations, and you could easily come to the conclusion that Continuous Delivery is all about speed, speed, speed. Sure, if you are releasing twice a year and your competitor is pushing new features every week, you will almost certainly be under pressure to figure out how you can deliver faster. By focusing too heavily on acceleration, however, we risk forgetting that there is no point dumping features into production in double-quick time if the system ends up breaking on a regular basis. Instead of talking about releasing software faster, we should be aiming to release software better...and time-to-delivery is only one of the metrics that we should be taking into account.

Of course, whenever the phrase "get features into production faster" is mentioned, there is usually an implicit assumption that we are making sure that the feature is actually working. It's true that testing is still often seen as the "poor cousin" of development, and QA as "the department of 'no'" rather than as essential members of the team. But pretty much every Continuous Delivery initiative that gets off the ground quickly realizes that automated testing—and, above and beyond that, early automated testing—is essential to building trust and confidence in the delivery pipeline.

In many cases, however, automated testing means code-level unit and integration testing, with only limited functional testing being carried out against a fully deployed, running instance of the system. Functional testing is obviously important to validate that whatever functionality is being added actually behaves in the intended fashion, and that related use cases "still work." However, as teams progress along the Continuous Delivery path of making smaller, more frequent releases with fewer functional changes, the emphasis of the pipeline's "quality control" activities needs to shift to follow suit: building confidence that the system will continue to behave and scale acceptably after the change is just as important—perhaps more important—than demonstrating that a small handful of new features work as planned.

It's at this point that automated functional testing proves insufficient, and we need to start adding automated stress and performance testing to the pipeline. More specifically: given the nature of today's highly distributed systems, we need to see a fully deployed, running instance of the system in action to see how its behavior has (or has not) changed in somewhat representative multi-user scenarios. As long as the system doesn't completely fall over, functional test results alone will not uncover problematic patterns that should stop our pipeline in its tracks.

To give but one real-world example: as part of a relatively minor feature change to only show the contents of a shopping cart if requested by the user, the data access logic was changed to load the item details lazily, and modified in a way that resulted in <# items in cart> calls to the database, rather than one. All functional tests still passed, since the impact of the additional database calls was not significant in the single- or low-user scenarios exercised by the functional tests. Without automated stress or performance testing in the pipeline, the change was released to production and quickly caused an expensive production outage.

Of course, adding automated stress and performance tests to a Continuous Delivery pipeline remains highly non-trivial. Even

with automated provisioning and deployment, spinning up an environment that is somewhat production-like, and getting a realistic configuration of your application deployed to it, is much more complicated than booting up a single instance of your app on a VM or in a container. Perhaps your organization maintains a dedicated performance lab, in which case the idea of "spinning up your own environment" is probably out of bounds right from the get-go. And we haven't even started talking about getting hold of sufficiently realistic data for your stress test environment yet. Given all that, how likely is it that we'll ever get this idea off the ground?

Here's the trick: we don't need almost-production-like environments for performance and stress tests to be useful in a Continuous Delivery pipeline. We don't even need the absolute response times of the multi-user tests we will be running—not if we are talking about a pipeline run for a relatively minor feature change for a system that's already in production, at any rate (of course, we'll want to get some reasonably realistic numbers for the first launch of a system, or before a major change that means that the current production instance is no longer a broadly accurate predictor).

What we need to do is run multi-user scenarios against a fully deployed instance of the system that is configured for scalable, multi-user use and compare the results against the previous pipeline runs. Whether a particular call takes 4s or 8s in our stress test environment is not that interesting—what matters is when a call whose 90th percentile has averaged 7.8s for the past couple of pipeline runs suddenly clocks in at 15.2s. In other words, deviations rather than absolute values are the canary in the coal mine that we are looking for.

How significant does the deviation have to be before we care? How do we avoid being flooded by false positives? To some extent, the answers to these questions are still largely down to the specifics of each individual system, and the team's choice. There are also a couple of statistical techniques we can apply here, such as setting the threshold for acceptable deviations in relation to the standard deviation, rather than allowing only a maximum percentage increase. We can also make use of outlier detection algorithms to flag values that are "way off the charts" and are more likely caused by a broken test environment than by the current change going through the pipeline. In addition, we can allow for re-calibration of our thresholds if we already know that the current change will impact the results for a specific scenario.

The classic metrics returned by stress or performance tests—response time and number of successful/failed calls—can be reasonably good at identifying changes to the system that may have a problematic impact. However, response time is influenced by a number of factors external to the system under test, which can be hard to control for, especially with on-demand testing environments: "noisy neighbor" VMs can influence performance; network latency can be impacted by other VMs in the same virtual segment; disk I/O can be variable; and so on.

Dedicated hardware is a possible solution, but it is expensive. One alternative is to run a small benchmarking suite on your on-demand environment and factor the underlying fluctuations in performance into the thresholds that trigger a pipeline failure. However, this is time-consuming and adds a complicated step to the processing of the test results. Happily, there is another option we can consider instead: measuring key architectural metrics directly.

This approach is based on the observation that many instances of poor performance and failure of large-scale systems are linked to a change in one of a relatively small number of metrics: number of calls to external systems (including databases); response time and size of those external calls; number of processing exceptions thrown (e.g. when an internal buffer or queue is full); CPU time taken; and other related metrics.

AS LONG AS THE SYSTEM DOESN'T COMPLETELY FALL OVER, FUNCTIONAL TEST RESULTS ALONE WILL NOT UNCOVER PROBLEMATIC PATTERNS THAT SHOULD STOP OUR PIPELINE IN ITS TRACKS.

Rather than keeping track only of performance testing results, we can, with suitable instrumentation and/or data collection tools, measure these architectural metrics directly. We can monitor their values over time and watch out for deviations in the same way as for response times. In fact, we can even consider collecting a subset of these metrics right at the beginning of our pipeline, when running code-level unit or integration tests! Some metrics, such as the amount of data returned from database calls, will not be of much use with the stub databases and sample datasets typically used for code-level testing. But we will still be able to spot a change that causes the system to jump from 1 to N+1 requests to the databases, as in our shopping cart scenario earlier.

In short, thinking of the purpose of our Continuous Delivery pipeline as delivering software better makes us focus on adding automated testing and metrics. The main purpose of these metrics is to give us confidence that the system will still continue to behave well, at scale, when the changes currently in the pipeline are added. An effective way to gather such measurements is to run performance or stress tests that simulate multi-user scenarios against a fully deployed, running instance of the system configured for scalable, multi-user use... without needing to incur the expense and complexity of building "fully production-like" environments.

Alongside the response time numbers returned by the performance tests, we can use log aggregators, APM tools, or custom instrumentation to track key architectural metrics inside the system. By looking for deviations in these key parameters, we can flag up pipeline runs that risk causing major system instability, allowing us to accelerate our delivery pipeline without putting quality at risk.

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.

CONTINUOUS DELIVERY TOOLS
by ThoughtWorks®

@snap_ci

www.snap-ci.com

BUILD, TEST, DEPLOY IN THE CLOUD

CLOSELY INTEGRATED WITH THE GITHUB WORKFLOW

AUTOMATE, TRACK AND VISUALIZE DEPLOYMENT

MULTI-STAGE PIPELINES

OPEN SOURCE CONTINUOUS DELIVERY SERVER

SUPPORTS BOTH SIMPLE AND COMPLEX WORKFLOW MODELING

DEPLOYMENT PIPELINES AS A CORE CONCEPT

UNIVERSAL PLATFORM AND TOOL SUPPORT

COMMERCIAL SUPPORT AND ENTERPRISE ADD-ONS, INCLUDING DISASTER RECOVERY

@goforcd

www.go.cd

3 Key Deployment Pipeline Patterns

Teams have been automating the build, test and deploy processes of their software for many years, but usually in a very specific "one off" manner. This piece walks through 3 key patterns, among many, to setting up a successful deployment pipeline.

BUILD THINGS ONCE

When you're taking software from initial code change to production in an automated fashion, it's important that you deploy the exact same thing you've tested.

Once you've built a binary, you should upload that back to your CD server or artifact repository for later use.

When you're ready to deploy to a downstream system, you should fetch and install that build artifact. This way you make sure that you're running your functional tests on the exact thing you built and that you're going to deploy.

VERIFY ON A PRODUCTION-LIKE ENVIRONMENT

Ideally you should be staging and testing on the same set up. If your staging and production environments are exactly the same, you can even use them interchangeably in a blue / green deployment pattern. It's hard to imagine a better way to be sure your software will work when you turn users loose on it.

If you're deploying to very large environments where it's just not practical to have an exact duplicate, you should still duplicate the technology. In other words, if you your web application runs on a cluster of 1,000 machines in production, you should verify that application on a cluster, even if it's only two machines.

One of the reasons people are so excited about advancements in container technology is the help they provide for this.

NON-LINEAR WORKFLOWS

It's not uncommon for teams to practice Continuous Integration on small parts of code before "throwing it over the wall" to someone else to put together with other parts of the application. In these cases, it's usually OK to set up a linear flow. The entire thing might run end to end in a just few minutes.

Don't feel like you need to emulate the often seen business diagram of pipelines that show unit-test, then functional tests, then security tests etc. The purpose of those diagrams is to show intent, not implementation. If you're able, run as much as possible at the same time for faster feedback.

WRITTEN BY KEN MUGRAGE
TECHNOLOGY EVANGELIST AT THOUGHTWORKS

GoCD BY THOUGHTWORKS

Open source continuous delivery server specializing in advanced workflow modeling.

CATEGORY	PRICE	OPEN SOURCE?
Continuous Delivery Management	Free. Commercial support and add-ons available.	Yes

STRENGTHS

- Universal platform and tool support
- Advanced pipeline dependency management that supports both simple and complex workflows
- Runs pipelines in parallel for faster feedback loops
- Tracks every change from commit to deploy

NOTABLE CUSTOMERS

- Coca-Cola
- Trainline
- Boston Scientific
- Ancestry
- Skype
- Baird

WEBSITE www.go.cd **TWITTER** @goforcd

Snap CI BY THOUGHTWORKS

Continuous delivery tool with deployment pipelines to build, test and deploy in the cloud.

CATEGORY	PRICE	OPEN SOURCE?
Continuous Delivery Management	Free 1-month trial	No

STRENGTHS

- Cloud based, zero infrastructure
- Closely integrated with GitHub workflow
- Allows deployment to AWS, Heroku and more
- Visibility through multi-stage pipelines
- Debug your build in the browser using snap-shell

NOTABLE CUSTOMERS

- BBC
- Escape the City
- Envato
- PlayOn! Sports
- Applauze
- Repairshopr

WEBSITE www.snap-ci.com **TWITTER** @snap_ci

How to Define Your Devops Roadmap

BY MIRCO HERING

QUICK VIEW

01 Defining the most appropriate DevOps roadmap depends on context.

02 In general, going wide (but not too wide) leads to longer lasting success and higher impact across the organization.

03 Adjust your roadmap to bring the organisation behind your efforts.

04 Don't get distracted by "shiny objects."

When adopting DevOps across the organization there are many different ways to set a path. You can take one application and go all the way to Continuous Delivery or you can take all your applications and just improve one area, like configuration management. Clearly there is not just one correct way – the silver bullet people are looking for. In this article, I will explore the parameters that help you define the path for your organization and provide some guidance.

CONTEXT IS KING

There is a general approach that I recommend for anyone defining a DevOps adoption strategy and roadmap, which I will describe further below. What people need to keep in mind is that their delivery context is different to everyone else's and that the adoption of "hot" practices or the implementation of new recommended tools will not guarantee you the results you are after. Far too often DevOps adoptions fall into one of two unsuccessful patterns:

FAILURE MODE 1 - Too little in the right place: Let's be honest with ourselves, the adoption of DevOps practices is never just an implementation project. Once you implement a specific practice like deployment automation or functional test automation, the real challenge is only just beginning. Truth be told, the 80/20 rule is a challenge for our efforts, as the last 20% of automation for deployments can be hardest and yet most benefits are dependent on removing that last manual step. Too often do I see the focus move off the early adoption areas, which then remain at the 80% mark and cannot reap the full benefits. This in turn can lead to deterioration of practices while maintenance efforts are high and yet the benefits are not outweighing it.

FAILURE MODE 2 - Local Optimization: Looking across the organizations that I have worked with, I see a disheartening pattern of local optimization. Some group goes off and implements DevOps practices (for example, the testing center of excellence or the operations team) and they have great successes. Their individual results improve so that the automated regression now runs in minutes rather than hours and giving developers a development environment from the cloud takes hours rather than weeks. Yet, the organization as a whole fails to see the benefits because the different practices are not compatible, or too many dependencies continue to hinder real results. Different test automation is run in the application test environment versus the one that runs in pre-production tests, and the cloud-based environment for developers is not compatible with the operations guidelines for production environments. Bringing these practices back together is costly and the associated change management is a nightmare.

HOW CAN YOU AVOID THESE FAILURE PATTERNS?

Follow these steps:

1. VISUALIZE YOUR DELIVERY PROCESS

Ask yourself if you really understand everything that is happening in your delivery process and how long it actually takes. More likely than not, you don't really know or are unsure. So, the first order of business is to bring stakeholders

from across the delivery lifecycle together and map the process out. Do this with all the stakeholders in the room. You as a process owner will be surprised what you can learn about your process from others who are exposed to your process or have to participate in it. Techniques from value stream mapping come in very handy here.

2. IDENTIFY PAIN POINTS AND BOTTLENECKS

Once you have the visualization of the process, add some quantifications on it, like how often feedback loops are run through, the volume of requests, and cycle times. You will use this information to identify pain points and bottlenecks. It will also help you with base-lining some performance metrics that you want to improve. It is good practice to define metrics for bottlenecks so that you can see them being removed.

3. ADDRESS THE MINIMUM VIABLE CLUSTER

Now to the secret sauce: Go wide, but just wide enough. Identify the minimum viable cluster that you can address and which will have significant meaning if you can improve it. The minimum viable cluster can be determined from your previous analysis of the process and the analysis of your application architecture as you are now able to identify a set of applications that needs to be delivered together. In most enterprise environments, applications are not independent (like they could be with microservices) and hence you need to address the integrated applications that change together. The pain points and bottlenecks will indicate which integration points are important (like those for performance testing, application deployments, or integration testing) while others with systems that don't change or have little data flow are less important. Make sure to challenge yourself on the minimal set as casting the net too wide will hinder your progress and being too narrow will mean you are unable to demonstrate the benefit. For this minimum viable cluster, you want to integrate as early in the lifecycle as possible by deploying all the required practices from the DevOps toolkit to get to short feedback cycles.

A key factor of success lies in the execution of your roadmap and the right governance. Making sure that you have tight feedback cycles and that the different groups remain aligned to the goals—as well as verifying the underlying technical architecture—is hard work. Enlist your change agents across the organization to help.

4. ADJUST YOUR ROADMAP TO BRING THE ORGANIZATION BEHIND YOUR EFFORTS

Everything I wrote so far points to a preference for going wide, yet there are very good reasons to break this pattern. I want to explore two of them in more detail:

ENABLING THE CHANGE: In some organizations there is skepticism that the change journey is going to be successful or in some cases there is not even sufficient knowledge to understand what "good" looks like. In those cases, it can be beneficial to showcase a particular application or technology. You usually choose a relatively isolated application that has suitable technology architecture to show how good delivery in a DevOps model looks like with minimum effort. The balance to be struck here is that this is meaningful enough to not be seen as "token effort" and yet it needs to be something where progress can be shown relatively quickly. I have seen internal portal applications or employee support systems, like a company mobile app, as great examples of this.

THE ADOPTION OF "HOT" PRACTICES OR THE IMPLEMENTATION OF NEW RECOMMENDED TOOLS WILL NOT GUARANTEE YOU THE RESULTS YOU ARE AFTER

CONSIDER YOUR STAKEHOLDERS: As with every other project in your organization, the support of your stakeholders is key. There will simply be no DevOps adoption program much longer if your supporters are getting fewer and fewer and the opponents are getting larger in number. When you look at your application portfolio, you might have applications which are not in the "minimum viable cluster" but which are of keen interest to some of your stakeholders. It is often worthwhile investing in those to make sure that they are ahead of the curve so that your stakeholders can "show off" their pet applications to the rest of the organization. The teams associated with those pet applications are often more open to pushing the boundaries as well.

DON'T GET DISTRACTED BY "SHINY OBJECTS"

It feels like no week is going by without a new DevOps product or concept being introduced to the community. As a technologist myself I understand the temptation to invest in a "quick Docker proof of concept," or "a microservice assessment of your portfolio." Yet, many organizations would benefit from focusing on the more basic practices of DevOps first. Often the benefits of those new products and concepts are only enabled when a certain maturity level is reached and, unfortunately, short cuts are not available in most cases. Make sure your efforts remain focused on the areas that will bring you benefits and don't chase hype until you are ready to really benefit from it.

In summary, you should go with a "go wide, but just wide enough" approach in most cases, as you want to demonstrate benefits to the organization. Those benefits will serve you during the next adoption phase and you can move ahead step by step on your adoption journey leveraging earlier benefits. The better you get at it, the wider and deeper you will be able to go.

MIRCO HERING leads Accenture's DevOps & Agile practice in Asia-Pacific, with a focus on Agile, DevOps, and CD, to establish lean IT organizations. He has over 10 years' experience in accelerating software delivery through innovative approaches and lately has focused on scaling these approaches to large complex environments. Mirco is a regular speaker at conferences and shares his insights on his blog.

Four Keys to Successful Continuous Delivery

The adoption of Agile practices in the last decade-and-a-half have greatly increased the ability of teams to produce quality software quickly. However, *producing* software is not the same as delivering value— many teams still struggle to deliver software quickly, relying on slow and often error-prone manual processes. Modern software teams must embrace Continuous Delivery if they are to compete in today's hyper-competitive market.

However, simply automating deployments is not enough. Continuous Delivery is part of a DevOps culture that prioritizes people and good processes, using technology as an enabler. Here are four keys to successful Continuous Delivery:

1. **EMBRACE A HOLISTIC DEVOPS CULTURE.** CD should be seen as part of a way of thinking that pervades every aspect of software delivery, from work management to development, testing, automated build and deployment to production monitoring.
2. **LEVERAGE THE POWER OF THE CLOUD.** Cloud computing allows teams to scale elastically, spin up environments on the fly and save costs through pay-as-you-use pricing. Even on-premises focused software can benefit from hosting QA environments in the Cloud or building hybrid services.
3. **CHOOSE OPEN AND EXTENSIBLE TOOLS.** Teams should prefer tools that are fit-for-purpose and extensible, so that they can be integrated into an end-to-end flow. Tools should always enable faster delivery and better quality.
4. **AUTOMATE, AUTOMATE, AUTOMATE!** Fast and reliable deliveries allow teams to shorten feedback cycles with more frequent deployments, focusing on solving business problems rather than plumbing.

Microsoft's Developer Division transformed Team Foundation Server (TFS) from a **Waterfall, two-year product release cycle** to an **Agile, cloud-cadence, 3-week release cycle** for Visual Studio Team Services (the SaaS version of TFS) and quarterly releases for the on-premises TFS product. Read Sam Guckenheimer's description of their journey and take the DevOps self-assessment at http://aka.ms/devops.

WRITTEN BY CLEMRI STEYN
GROUP PRODUCT MARKETING MANAGER, MICROSOFT

A SCORECARD FOR MEASURING ROI OF A

Continuous Delivery Investment

DevOps metrics consider both the quality of the software and the speed with which it's delivered to customers. This is all well and good in principle, but how does an improvement in agility translate to a business case that will persuade leadership to adopt Continuous Delivery?

Use calculations in this scorecard to estimate ROI from adopting specific DevOps practices. The aggregate sum of per-practice ROIs will make the strongest overall argument, but don't miss the trees for the forest.

Constants included in each formula are approximate and based on empirical research cited in bit.ly/CDROI

Revenue Gains From Accelerated Time to Market of New Functionalities (GTM)

_____GTM =

(revenue per year) x 115/100 (revenue increase estimate)

Gains From Flexibility in the IT Environment (GFX)

_____GFX =

(application annual revenue) x 0.036 (average TCO reduction x cost of IT as a percentage of revenue)

Gains From Cost Reduction of Application Failures Resulting From Increased Quality (GQL)

_____GQL =

(failures per year) x 28.3 minutes (average minutes to recover difference) x (revenue per minute)

Gains From Enhanced IT Team Productivity and Cost Reduction of IT Headcount Waste (GHC)

_____GHC (IT Ops) =

(IT average salary) x (IT staff headcount) x 0.16 hours(saved hours per week / total hours)

_____GHC (Developers) =

(developer average salary) x (developer staff headcount) x 0.18 hours ([time reduction by implementing continuous delivery] x [time spent on root cause analysis] x [time developers spend on problem resolution])

$$\frac{(GTM + GFX + GQL + GHC) - \text{Cost of Investment} \times 100}{\text{Cost of Investment}} = \textbf{Total Continuous Delivery ROI}$$

Used with permission by Oda Benmoshe and Zend Technologies. Original source located here: **bit.ly/CDROI**

They say software releases can be painless.

Congratulations if you've reached that celestial state. For the rest of us, releases are stressful at best, excruciating at worst. But which parts of the release process are most likely to cause pain? We did a survey to find out.

2233 developers told us where they feel pain during release. **Here's what they said.**

SURVEY BY DZONE; ANALYSIS BY DARÍO MACCHI

Continuous Delivery & Release Automation for Microservices

BY ANDERS WALLGREN

QUICK VIEW

01
Microservices introduce architectural complexities and management overhead—particularly on the testing and ops side.

02
Your goal should be to automate the releases of microservices-driven apps so they are reliable, repeatable, and as painless as possible.

03
For both CD and microservices, a highly-focused, streamlined automation pipeline is critical.

As software organizations continue to invest in achieving Continuous Delivery (CD) of their applications, we see increased interest in microservices architectures, which—on the face of it—seem like a natural fit for enabling CD.

In microservices (or its predecessor, SOA), the business functionality is decomposed into a set of independent, self-contained services that communicate with each other via an API. Each of the services has their own application release cycle, and are developed and deployed independently—often using different languages, technology stacks and tools that best fit the job.

By splitting the monolithic application into smaller services and decoupling interdependencies (between apps, dev teams, technologies, environments, and tooling), microservices allow for more flexibility and agility as you scale your organization's productivity.

While things may move faster on the dev side, microservices do introduce architectural complexities and management overhead—particularly on the testing and ops side. What was once one application, with self-contained processes, is now a complex set of orchestrated services that connect via the network. This has impact on your automated testing, monitoring, governance and compliance of all the disparate apps, and more.

A key prerequisite for achieving Continuous Delivery is automating your entire pipeline—from code check-in, through build, test, and deployments across the different environments—all the way to the production release. In order to support better manageability of this complex process, it's important to leverage a platform that can serve as a layer above any infrastructure or specific tool/technology and enable centralized management and orchestration of your toolchain, environments and applications. I'd like to focus on some of the implications microservices have on your pipeline(s), and some best practices for enabling CD for your microservices-driven application.

THE CHALLENGES OF MANAGING DELIVERY PIPELINES FOR MICROSERVICES

THE MONO/MICRO HYBRID STATE

It's very hard to design for microservices from scratch. As you're starting with microservices (as a tip: only look into microservices if you already have solid CI, test, and deployment automation), the recommendation is to start with a monolithic application, then gradually carve out its different functions into separate services. Keep in mind that you'll likely need to support this Mono/Micro hybrid state for a while. This is particularly true if you're re-architecting

a legacy application or are working for an organization with established processes and requirements for ensuring security and regulatory compliance.

As you re-architect your application, you will also need to architect your delivery pipeline to support CD (I'll expand more on this later on). It's important that your DevOps processes and automation be able to handle and keep track of both the "traditional" (more long-term) application release processes of the monolith, as well as the smaller-batch microservices/CD releases. Furthermore, you need to be able to manage multiple microservices—both independently and as a collection—to enable not only Continuous Delivery of each separate service, but of the entire offering.

INCREASE IN PIPELINE VARIATIONS

One of the key benefits of microservices is that they give developers more freedom to choose the best language or technology to get the job done. For example, your shopping cart might be written in Java, but the enterprise messaging bus uses Erlang. While this enables developers to "go fast" and encourages team ownership, the multitude of services and the different technologies that your pipeline automation would need to support grows considerably.

This need for flexibility creates challenges in terms of the complexity of your pipeline, its reusability, and its repeatability. How do you maximize your efforts and reuse established automation workflows across different technologies and tools?

ENSURING GOVERNANCE AND COMPLIANCE

With so many independent teams and services, and the diversity of tools and processes, large organizations struggle to standardize delivery pipelines and release approval processes to bring microservices into the fold with regards to security, compliance and auditability.

How do you verify that all your microservices are in compliance? If there's a breach or failure, which service is the culprit? How do you keep track of who checked-in what, to which environment, for which service, and under whose approval? How do you pass your next audit?

INTEGRATION TESTING BETWEEN SERVICES BECOMES MORE COMPLEX

When testing a service in isolation, things are fairly simple: You do unit testing and verify that you support the APIs that you expose. Testing of microservices is more complicated, and requires more coordination. You need to decide how you're handling downstream testing with other services: Do you test against the versions of the other services that are currently in production? Do you test against the latest versions of the other services that are not yet in production? Your pipeline needs to allow you to coordinate between services to make sure that you don't test against a version of the service that is about to become obsolete.

SUPPORTING THE PROLIFERATION OF HETEROGENEOUS ENVIRONMENTS

Microservices often result in a spike in deployments that you now need to manage. This is caused by the independent deployment pipelines for each service across different stacks or environments, an increase in the number of environments throughout the pipeline, and the need to employ modern deployment patterns such as Blue/Green, Canary, etc.

ONLY LOOK INTO MICROSERVICES IF YOU ALREADY HAVE SOLID CI, TEST, AND DEPLOYMENT AUTOMATION

Deployment automation is one of the key prerequisites for CD—and microservices require that you do a lot of it. You don't only need to support the volume of deployments; your pipeline must also verify that the environment and version are compatible, that no connected services are affected, and that the infrastructure is properly managed and monitored. While not ideal, at times you will need to run multiple versions of your service simultaneously, so that if a service requires a certain version and is not compatible with the newer one, it can continue to operate (if you're not always backwards-compatible).

In addition, microservices seem to lend themselves well to Docker and container technologies. As dev teams become more independent and deploy their services in a container, Ops teams are challenged to manage the sprawl of containers, and to have visibility into what exactly goes on inside that box.

SYSTEM-LEVEL VIEW AND RELEASE MANAGEMENT

System-level visibility is critical, not only for compliance, but also for effective release management on both the technical and business sides. With complex releases for today's microservices-driven apps, you need a single pane of glass into the real-time status of the entire path of the application release process. That way, you ensure you're on schedule, on budget, and with the right set of functionality. Knowing your shopping-cart service will be delivered on time does you no good, if you can't also easily pinpoint the status of the critical ERP service and all related apps that are required for launch.

BEST PRACTICES FOR DESIGNING CD PIPELINES FOR MICROSERVICES

You want to embrace microservices as a means to scale and release updates more frequently, while giving operations

people the platform to not only support developers, but to also operationalize the code in production and be able to manage it. Because microservices are so fragmented, it is more difficult to track and manage all the independent, yet interconnected components of the app. Your goal should be to automate the releases of microservices-driven apps so these are reliable, repeatable, and as painless as possible.

WHEN CONSTRUCTING YOUR PIPELINE, KEEP THESE THINGS IN MIND:

1. Use one repository per service. This isolation will reduce the engineer's ability to cross-populate code into different services.
2. Each service should have independent CI and Deployment pipelines so you can independently build, verify, and deploy. This will make set-up easier, requires less tool integration, provides faster feedback, and requires less testing.
3. Plug-in all your tool chain into your DevOps Automation platform—so you can orchestrate and automate all the things: CI, testing, configuration, infrastructure provisioning, deployments, application release processes, and production feedback loops.
4. Your solution must be tools/environment agnostic—so you can support each team's workflow and tool chain, no matter what they are.
5. Your solution needs to be flexible to support any workflow—from the simplest two-step web front-end deployment to the most complex ones (such as in the case of a complex testing matrix, or embedded software processes).
6. Your system needs to scale to serve the thousands of services and a multitude of pipelines.
7. Continuous Delivery and microservices require a fair amount of testing to ensure quality. Make sure your automation platform integrates with all of your test automation tools and service virtualization.
8. Auditability needs to be built into your pipeline automatically so you always record in the background the log of each artifact as it makes its way through the pipeline. You also need to know who checked in the code, what tests were run, pass/fail results, on which environment it was deployed, which configuration was used, who approved it, and so on.
9. Your automation platform needs to enable you to normalize your pipelines as much as possible. Therefore, use parameters and modeling of the applications/environment and pipeline processes so you can reuse pipeline models and processes between services/teams. To enable reusability, the planning of your release pipeline (and any configuration or modeling) should be offered via a unified UI.
10. Bake compliance into the pipeline by binding certain security checks and acceptance tests, and use infrastructure services to promote a particular service through the pipeline.
11. Allow for both automatic and manual approval gates to support regulatory requirements or general governance processes.
12. Your solution should provide a real-time view of all the pipelines' statuses and any dependencies or exceptions.
13. 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.

For both Continuous Delivery and microservices, a highly-focused and streamlined automation pipeline is critical to reduce bottlenecks, mitigate risk, and improve quality and time-to-market. While some may choose to cobble together a DIY pipeline, many organizations have opted for a DevOps automation or Continuous Delivery platform that can automate and orchestrate the entire end-to-end software delivery pipeline. This is particularly useful as you scale to support the complexities of multitudes of microservices and technologies. You don't build your own email server, so why build this yourself?

WANT MORE?

For more tips on microservices, to learn if it is right for you and how to start decomposing your monolith, check out this video of my talk on "Microservices: Patterns and Processes" from the recent DevOps Enterprise Summit.

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. He invites you to download the community edition of ElectricFlow to build your pipelines and deploy any application, to any environment—for free.

DIVING DEEPER

INTO CONTINUOUS DELIVERY

TOP 10 #CD TWITTER FEEDS

 @MARTINFOWLER

 @JEZHUMBLE

 @TESTOBSESSED

 @DAMONEDWARDS

@DOESSUMMIT

 @ALLSPAW

 @ADRIANCO

 @ERICMINICK

 @ARRESTEDDEVOPS

 @KARTAR

DZONE CD-RELATED ZONES

DevOps Zone

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/system admins. The DevOps Zone is your hot spot for news and resources about Continuous Delivery, Puppet, Chef, Jenkins, and much more.

Agile Zone

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 specific techniques, the core principles of the Agile Manifesto can help any organization in any industry to improve their productivity and overall success. Agile Zone is your essential hub for Scrum, XP, Kanban, Lean Startup and more.

Cloud Zone

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. This Zone focuses on PaaS, infrastructures, security, scalability, and hosting servers.

TOP CD REFCARDZ

Continuous Delivery Patterns

Deployment Automation Patterns

Getting Started With Docker

TOP CD WEBSITES

DevOps devops.com

Script Rock scriptrock.com

DevOpsGuys blog.devopsguys.com

TOP CD RESOURCES

Puppet Labs State of DevOps Report

The Phoenix Project
by Gene Kim and Kevin Behr

Continuous Delivery
by Jez Humble and Dave Farley

THE CONTINUOUS DELIVERY Maturity Checklist

Continuous Delivery is about making software provide more value in the real world. And your CD process should itself continuously improve.

This checklist will help you set delivery goals and find ways to deliver software more effectively at both technical and organizational levels.

TO USE: simply divide the number of items you checked off by 47 to get your maturity score as a percent.

ALIGNMENT

"Unifying group and individual direction and goals around the singular vision of the organization."

- ☐ We prioritize according to business objectives.
- ☐ We volunteer for tasks rather than having them assigned.
- ☐ Our team has clear objectives that correspond with our company vision.
- ☐ Our product team is focused on sustainable velocity rather than current speed.
- ☐ We focus on time to repair rather than time between issues.
- ☐ DevOps is not isolated to a specific role in our organization.
- ☐ DevOps is not isolated to a specific team in our organization.
- ☐ Our operational functions are seen as a source of competitive advantage.

CONTEXT

"Making relevant information and contact available to those who need it, when they need it."

- ☐ Representation from our operations team is involved in development sprint planning.
- ☐ We make potential changes visible to all members of our product team.
- ☐ We have an automated system for running tasks and receiving notifications with our team chat.
- ☐ We consult with auditors and regulators regularly and seek guidance when designing systems.
- ☐ Our team is encouraged to question tasks and priorities.
- ☐ We have a centralized instant message system including all members of our product team.
- ☐ All members of our product team have access to environment status, metrics, and history.
- ☐ All members of our product team have access to code status, metrics, and history.

LEARNING

"Empowering personal growth and fostering understanding through continuous improvement."

- ☐ We cultivate an environment of continuous learning.
- ☐ We regularly celebrate our product team's learnings and successes internally.
- ☐ We regularly share our product team's learnings and successes with the rest of our organization.
- ☐ We openly discuss failures in order to share learning.
- ☐ We identify skills needed to improve or address future objectives.
- ☐ We strive to examine how we complete our work, and how effectively we complete it.
- ☐ We estimate based on measurement and past experience.

LIFECYCLE

"Focus on software as a product deserving of care, attention, and reflection, within an ever-changing ecosystem."

- ☐ Our software development cycle is 2 weeks or less.
- ☐ Our software development cycle is defined by releasing a working change into production.
- ☐ We stop development upon discovering a defect and prioritize its repair.
- ☐ Developers or product owners are able to deploy our product to production.
- ☐ We have automated testing prior to automated production deployment.
- ☐ Our configuration of systems is automated.
- ☐ Our deployed system configuration is immutable.
- ☐ Our release and deployment automation is environment agnostic.

ORGANIZATION

"Providing structure for interaction and cohesion supporting collaboration and productivity."

- ☐ Our subject matter expertise is not isolated to individuals.
- ☐ We enable peer and cross-functional review for changes.
- ☐ Our organization is configured around cross-functional teams.
- ☐ Our teams are customer and product oriented.
- ☐ We review priorities on a regular basis.
- ☐ Our developers have access to production-like systems to work and test on.
- ☐ Our developers have access to production-like data to work and test against.
- ☐ Our developers have access to dependencies required to build and test software.

PROCESS

"Rituals crafted to foster consistency and confidence, providing a framework for continuous improvement."

- ☐ Our organization follows agile development practices.
- ☐ We practice blameless postmortems.
- ☐ We regularly examine constraints in our delivery process.
- ☐ Our system configuration is committed into version control.
- ☐ Our documentation is version controlled and shared.
- ☐ We maintain a backlog of tasks, visible to all team members and available for comment.
- ☐ We practice test- or behavior-driven development.
- ☐ We test our changes against a merge with our mainline code.
- ☐ We test our changes against production-equivalent load and use patterns.

Used with permission by Steve Pereira. Check out the interactive version of this checklist at devopschecklist.com.

The Benefits of Database Continuous Delivery

By Grant Fritchey

I spend a lot of time talking about the need for automation within my database, especially in support of development, deployment, and through-process management. It's because I learned the hard way just how important it is. It took a pretty headstrong developer to convince me that I was doing database development the wrong way. However, once he made me see the light, I felt like one of the Blues Brothers - on a mission.

I think this is one of the hardest parts to get people to understand: if you have a mostly, or completely, manual database deployment process, you're experiencing pain from the inefficiencies that causes. However, it's a pain that's just embedded into the organization so much, that you either don't notice it any more, or you don't believe that the pain can go away. It really can. But, don't listen to me.

Here's an excellent discussion from Skyscanner about how they went from deploying once every 4-6 weeks to 95 times a day. If you don't think that radically reduces pain within the organization, let alone makes them more agile in support of the business... well, you're wrong.

One thing you'll have discovered early on in your Continuous Delivery efforts is that automation is literally essential. Everything that a computer can do faster and more reliably than a human must be automated in the name of tight, robust feedback cycles and faster learning. Speed is actually just a wonderful byproduct of having automated and reliable processes that help you improve with every iteration, which is what Continuous Delivery and Database Lifecycle Management are fundamentally about.

From the data gathered and compiled for this guide, we can see that nearly two-thirds of DZone readers have implemented Continuous Delivery for their application build management, and nearly everyone else wants to.

> "What we're really talking about in many parts of the industry is culture change, and that's always the most difficult thing for an organization to adopt. I think that there IS a need to refresh the culture in a few areas of Database Lifecycle Management."
>
> **Dave Farley**
> **Co-author of Continuous Delivery**

This is great progress for software development as a craft. However, less than one-third of those people have implemented Continuous Delivery for their databases, and those are the foundations that your software is built on.

The good news is that it's easier than ever to include your database in your Continuous Delivery pipeline, and the results speak for themselves. But hey, that's just me and one other organization... well, and Fitness First, the largest privately-owned health group in the world, who are changing their deployments from a cycle of months to a cycle of days. They're moving at speed, they're delivering software quickly for the business, and that's what it's all about.

Oh yeah, and Yorkshire Water, one of the UK's major utility companies, who are talking about the time and money they're saving because of their deployment automation.

In short, this is a thing that many organizations are doing, and you can as well. Your manual processes are taking time, causing outages, and preventing you from moving quickly in support of your business. You absolutely need to get on the Database Lifecycle Management / Application Lifecycle Management / DevOps / Continuous Delivery band wagons. We're changing how we develop and deploy databases and software in order to move with the business. Come on in, the water is fine.

Oh, and if you really do want to dive into DLM, I'm working on a new book that we're delivering using an agile process. You can check it out here.

> "If you're not including your database in your Continuous Delivery process, then you're not really doing Continuous Delivery."

Continuous Delivery for Containerized Applications

QUICK VIEW

01 Containers are a key enabler of rapid deployment and delivery cycles.

02 Container management tools must provide governance and visibility across each phase of a deployment pipeline.

03 Containers make it easier to adopt microservices and CD best practices like blue-green deployments and canary launches for testing in production.

04 With the proper tooling, entire environments can now be created on demand, allowing new efficiencies for development and testing.

BY JIM BUGWADIA

Application containers, like Docker, are enabling new levels of DevOps productivity by enabling best practices, like immutable images, and allowing proper separation of concerns across DevOps and platform teams. In this article, we will discuss how containers make it easier to fully automate a continuous delivery pipeline, from a commit to running code in production environments. We will also examine best practices for defining and managing containers across a CI/CD pipeline, as well as best practices for deploying, testing, and releasing new application features.

CONTAINER IMAGES AND TAGS

The lifecycle of an application container spans development and operations, and the container image is a contract between development and operations. In a typical cycle, code is updated, unit tested, and then built into a container image during the development phase. The container image can then be pushed into a central repository. Next, while performing tests or deploying the application, the container image can be pulled from the central repository.

Since the image can be updated several times, changes need to be versioned and managed in an efficient way. For example, Docker images use layers and copy-on-write semantics to push and pull only the updated portions of an image.

In Docker terminology, container images are stored in an Image Registry, or a registry (e.g. Docker Hub, Google Container Registry, Quay, etc.). Within a registry, each application container has its own Image Repository, which can contain multiple tags.

Docker allows multiple tags to be applied to a container image. Think of tags as a named pointer to an image ID. Tags provide the basic primitive for managing container images across the delivery pipeline.

As of today, Docker tags are mutable. This means that a tag can point to a different image over time. For example, the tag "latest" is commonly used to refer to the latest available image. While it's convenient to be able to change which image a tag points to, this also poses a problem where pulling a tag does not guarantee getting the same image.

There are pending requests from the community to introduce the immutable tags as a feature in Docker, or to provide the ability to pull an image using the image ID, which is immutable. Until these are addressed, a best practice is to automate the management of tags and to establish a strict naming convention that separates mutable from immutable tags.

BUILDING IMMUTABLE CONTAINER IMAGES

When using application containers, a developer will typically write code and run unit tests locally on their laptop. The developer may also build container images, but these are not ready to be consumed by other team members, and so will not be pushed to the Image Registry.

A best practice is to maintain the automated steps to containerize the application as part of the code repository. For Docker these steps are defined in a Dockerfile [2], which can be checked in alongside the code. When changes are committed a build orchestration tool, like Jenkins or Bamboo, can build and tag the container image, and then push the image to a shared Image Registry.

With this approach, each build creates an immutable image for your application component, which packages everything necessary to run the component on any system that can host containers. The image should not require any additional configuration management or installation steps. While it may seem wasteful to create an immutable image with every build, container engines like Docker optimize image management, using techniques like copy-on-write, such that only the changes across builds are actually updated.

Even though the application component does not need to be re-configured each time it is deployed, there may be configuration data that is necessary to run the component. This configuration is best externalized and injected into the runtime for the container. Container deployment and operations tools should allow injecting configuration as environment variables, dynamically assign any bindings for storage and networking, and also dynamically inject configuration for services that are dependent on each other. For example, while creating an environment in Nirmata you can get environment variables to one or more services.

CONTAINER-AWARE DEPLOYMENT PIPELINES

A deployment pipeline consists of various steps that need to be performed to build, test, and release code to production. The steps can be organized in stages, and stages may be fully automated or require manual steps.

Once you start using application containers, your deployment pipeline needs to be aware of container images and tags. It is important to know at which phase of the deployment pipeline a container image is. This can be done as follows:

1. Identify the stages and environment types in your pipeline
2. Define a naming convention for immutable image tags that are applied to each image that is built by the build tool. This tag should never be changed:

   e.g. {YYYYMMDD}_{build number}
3. Define naming conventions for image tags that will be accepted into an environment:

   e.g. {environment name}_latest
4. Define naming conventions for image tags that will be promoted from an environment to the next stage in the deployment pipeline:

   e.g. {next environment name}_latest

Using these rules, each container image can have at least two tags, which are used to identify and track progress of a container image across a deployment pipeline:

1. A unique immutable tag that is applied when the image is built and is never changed
2. A mutable tag that identifies the stage of the image in the deployment pipeline

Application container delivery and lifecycle management tools can now use this information to govern and manage an automated deployment pipeline. For example, in Nirmata you can define environment types that represent each phase in your deployment pipeline. A tag naming scheme is used to identify which tags are allowed into each environment type, and how to name tags for images that are promoted from an environment type.

UPDATING APPLICATIONS CONTAINERS

So far, we have discussed how to build container images and manage container images across a deployment pipeline. The next step is to update the application in one or more environments. In this section, we will discuss how containers ease adoption of best practices for updating applications across environments.

MICROSERVICES

"Microservices" refers to an architectural style where an application is composed of several independent services, and each service is designed to be elastic, resilient, composable, minimal, and complete [3]. Microservices enable agility at scale, as organizations and software code bases grow, and are becoming increasing popular as an architectural choice for enterprise applications.

ONE OF THE BENEFITS OF A MICROSERVICES STYLE APPLICATION IS GRANULAR VERSIONING AND RELEASE MANAGEMENT, SO THAT EACH SERVICE CAN BE VERSIONED AND UPDATED INDEPENDENTLY."

Containers are fast to deploy and run. Due to their lightweight packaging and system characteristics, containers are the ideal delivery vehicle for microservices, where each individual service has its own container image and each instance if the service can now run in its own container.

One of the benefits of a Microservices-style application is granular versioning and release management, so that each service can be versioned and updated independently. With the Microservices approach, rather than testing

and re-deploying the entire system with a large batch of changes, small incremental changes can be safely made to a production system. With the proper tooling, it is also possible to run multiple versions of the same service and manage requests across different versions of the service.

BLUE-GREEN DEPLOYMENTS

A blue-green deployment (sometimes also called red-black deployment) is a release management best practice that allows fast recovery in case of potential issues [4]. When performing a blue-green update, a new version ("green") is rolled out alongside an existing version ("blue") and an upstream load balancer, or DNS service, is updated to start sending traffic to the "green" version. The advantage of this style of deployment is that if something fails you simply revert back the traffic to the "blue" version, which is still running as a standby.

Containers make blue-green deployments faster and easier to perform. Since containers provide immutable images, it is always possible to revert to a prior image version. Due to the optimized image management capabilities, this can be done in a few seconds.

However, the real value of containers comes through as you start combining blue-green deployments with microservices-style applications. Now individual services can leverage this best practice, which further helps reduce the scope of changes and potential errors.

CANARY LAUNCHES

A canary launch goes a step further than blue-green deployments and provides even greater safety in deploying changes to a production system [5]. While with blue-green deployments, users are typically using either the blue or the green version of the application component, with a canary launch the new version runs alongside the older versions and only select users are exposed to the new version. This allows verifying the correct operation of the new version, before additional users are exposed to it.

For example, you can upgrade a service to a new version (v6.1, for instance) and initially only allow calls from internal or test users to that service. When the new version of the service looks stable, a small percentage of production traffic can be directed to the new version. Over time, the percentage or production traffic can be increased, and the old version is decommissioned.

While containers are not necessary for implementing and managing canary launches, they can be an enabler in standardizing and automating update policies. For example, Nirmata allows users to select on a per-environment basis how to handle service updates. Users can choose to simply be notified and manually trigger a rolling upgrade, can choose to add the new version alongside existing versions, or can choose to replace existing versions via a rolling upgrade.

ENVIRONMENTS ARE NOW DISPOSABLE

Cloud computing has enabled software-defined infrastructure, and allows us to treat servers as disposable entities [6]. Containers take this a step further. Containers are very fast to deploy and launch, and with the right orchestration and automation tools you can now treat entire environments as on-demand and disposable entities.

WHILE CONTAINERS SOLVE SEVERAL KEY PROBLEMS, THEY ALSO REQUIRE NEW TOOLING FOR AUTOMATION AND MANAGEMENT OF APPLICATIONS.

While production environments are likely to be long-lived, this approach provides several benefits for development and test environments, which can now be recreated with a single click. A deployment pipeline can now incorporate automated tests, which spin up environments, run tests, and if the tests succeed dispose of the environment.

SUMMARY

Containers are rapidly being embraced as a foundational tool for DevOps and continuous delivery. Along with Microservices-style architectures, containers enable and even help enforce best practices for managing application components across a delivery pipeline, from a commit to running in production.

While containers solve several key problems, they also require new tooling for automation and management of applications. Next generation application delivery and management solutions, such as Nirmata, can now leverage containers as a standardized building block in order to fully automate the application lifecycle. This combination of technologies will help unleash a new level of productivity and advance software engineering to fulfill the ever-growing need for software-enabled products and devices across all domains.

[1] docs.docker.com/engine/introduction/understanding-docker
[2] docs.docker.com/v1.8/reference/builder
[3] nirmata.com/2015/02/microservices-five-architectural-constraints
[4] martinfowler.com/bliki/BlueGreenDeployment.html
[5] infoq.com/news/2013/03/canary-release-improve-quality
[6] slideshare.net/randybias/pets-vs-cattle-the-elastic-cloud-story

JIM BUGWADIA started his career developing C++ software at Motorola for cellular network infrastructure, where his team launched the world's first CDMA network. Since then Jim has held several developer, architect, and leadership roles at companies like Bell Labs, Cisco, Trapeze Networks, and Pano Logic. In 2013 Jim founded Nirmata, a startup focused on solving Enterprise DevOps pain points, by providing multi-cloud management of Microservices style applications that are deployed using containers.

Compete or Die:

How Silicon Valley Achieves Developer Productivity

COMPETE OR DIE

Software is eating the phone and car industry, and the world. Elite ***Silicon Valley teams*** are raising the competitive bar with unheard-of-levels of productivity at scale. Meanwhile your team faces a ***crisis of complexity*** driven by open-source dependencies, platform fragmentation, polyglot, complex release variation, containers, and DevOps. Today, it's ***compete or die.***

INCOMING BREAKING CHANGES: THE WALL OF DEATH

As you add developers, incremental developer productivity tends towards zero. Scale can slow release cadence and even your best developers can drown in debugging the logjam of ***incoming breaking changes*** (from changes in other people's code, or from complex combinations of changes). The "***Wall of Death***" is reached when 100% of developer time is spent on debugging incoming changes and 0% on new features. Adding developers becomes counterproductive.

In addition to the speed of incoming changes, the two key variables that govern the "Wall of Death" are the percent of inbound changes that break and the insight (and speed) in resolving breaks. Gradle provides developers the best control over these.

WHAT SILICON VALLEY IS DOING

Linkedin, Netflix, Apple, Palantir, Google, and ***Unity Technologies*** are using Gradle to move the Wall of Death and achieve unprecedented speed at scale. With over ***10M downloads*** of Gradle in 2015 (doubling yearly), the momentum behind Gradle is unstoppable and the benefits significant. For example, Netflix uses Gradle to ship continuously (multiple times a day, every day) and has seen build times as long as 80 minutes drop to less than 5 minutes. Gradle is free and open source software, and combined with Gradle.com, SaaS can move the Wall of Death by reducing build times and release intervals by ***orders of magnitude***. Gradle also has 700+ community plugins that integrate every imaginable DevOps tool. Learn more at gradle.org.

WRITTEN BY MIKO MATSUMURA
CHIEF MARKETING OFFICER AT GRADLE INC

Gradle BY GRADLE INC.

Gradle is the heart of CD for the best teams in the world. Free Open Source Sofware integrated with every other tool in this guide and popular developer IDEs.

CATEGORY
Build Automation

NEW RELEASES
3.0 coming soon

OPEN SOURCE?
Apache 2 License

CASE STUDY

Want to ship software 30x faster at massive scale? LinkedIn achieved this with Gradle by going from a 30 day release schedule to multiple times every day. They automate builds in 42 languages from Java, C++, Python, to Hadoop using Gradle and the 2,000+ developers who actively use Gradle generate over 300,000 Gradle builds per week. This Enterprise-wide Gradle deployment contains over 4000 submodules. Jens Pillgram Larsen, head of the developer tools team at Linkedin said "Build Automation at Linkedin is synonymous with Gradle." The best software teams in the world including Netflix, Apple, Palantir, Google, Nike and many more are making deep investments in Gradle. Learn more at gradle.org or gradle.org/gradle-in-the-enterprise-at-linkedin

STRENGTHS

- Used by the best teams in the world
- Free, Open Source Software
- Popular (10M+ downloads in 2015)
- Deep IDE integration
- 700+ Plugins to every major DevOps tool
- Powerful yet concise build scripts
- Developer friendly wrapper deployment
- Highly extensible and customizable

NOTABLE CUSTOMERS

- Netflix
- LinkedIn
- Palantir
- Unity Technologies
- Apple
- Nike
- Google

BLOG gradle.org/blog | **TWITTER** @gradle | **WEBSITE** www.gradle.org

How to Test Your CI Server for Scalability

Ever wondered what your CI server is capable of?

One of the ways to measure your CI server scalability is to test the number of agents, or "build slaves," the server can hold while all of the agents are running concurrent tasks. We at JetBrains believe it's an important measure of success because, ultimately, increasing the number of build agents increases the number of builds you can run simultaneously—provided your server is able to handle all of them.

Testing will definitely depend on your CI server architecture. We tested TeamCity—the CI server we build and use ourselves – using the Benchmark plugin we created internally.

TeamCity comprises a server and a number of agents. Agents run builds and send results (build log, artifacts, tests, etc.) back to the server. Most of the time the results are sent in real time while the build is still running, which allows for displaying a nice real-time progress of the build in the web interface, for sending notifications early, and more.

Thus, to benchmark the server we do the following:

- Generate build logs from a lot of agents for some time with a constant rate
- Stop the load generator and measure the amount of build logs processed on the server up to this time
- Once we know the duration of the benchmark and the rate of build log generation, we can estimate the amount of build logs which should have been processed
- Having this delta (estimated amount – actual amount) we can judge whether the server is OK with the load

We conducted the above mentioned benchmark for TeamCity 9.1, and it showed TeamCity to work well with up to 300 build agents (300 concurrently running builds actively logging build runtime data). In synthetic tests the server was functioning okay with as many as 500 agents (the server with 8 cores, 32Gb of total memory running under Linux, and MySQL server running on a separate comparable machine). Try benchmarking your CI server and see what it's capable of.

WRITTEN BY YEGOR NAUMOV
PRODUCT MARKETING MANAGER AT JETBRAINS

TeamCity BY JETBRAINS

TeamCity is a free continuous integration and deployment server. It provides out-of-the-box continuous build and unit testing with early reporting of found problems, code quality analysis, deployment routines, and simple server administration and maintenance.

CATEGORY
Continuous Integration and Delivery

NEW RELEASES
Twice a year

OPEN SOURCE?
No

CASE STUDY

RecruitmentGenius.com, the UK's No. 1 online recruitment agency, has taken advantage of TeamCity while aggressively developing their online platform. TeamCity was installed and configured to automatically build their solution every time a change was made. Additional build steps for finding code duplicates and running code coverage reports on unit test run and build features were configured, which helped improve the overall quality of the solution. They utilized the Automatic Merge build feature for Git to ensure that changes are built on a private branch, and those changes are merged into the main branch only if the build finishes successfully. This ensures that the main branch is always in an unbroken state.

STRENGTHS

- Simple project setup from a VCS URL
- On-the-fly build progress reporting for finding out about failed tests immediately
- Templates for creating any number of jobs
- Build chains and dependencies for breaking down build procedures
- Stored project settings in VCS

NOTABLE CUSTOMERS

- HP
- eBay
- Tesla
- Nvidia
- Boeing
- Nintendo

BLOG blog.jetbrains.com | **TWITTER** @teamcity | **WEBSITE** jetbrains.com/teamcity

Securing a Continuous Delivery Pipeline

BY MATTHEW SKELTON & MANUEL PAIS

QUICK VIEW

01
With a collaborative mindset, security officers, developers, and testers can come together and help each other with their respective gaps in knowledge and experience.

02
At an early stage it might be advisable to keep manual security controls in the delivery pipeline.

03
Tools and practices alone don't lead to more secure systems; we also need frequent and open communication, feedback, and empathy for other people.

Security officers have an acute sense of duty. A duty to protect the organization from malicious attacks; to find holes in the organization's software and infrastructure assets and track their resolution. That often requires saying no to new release deployments, refusing to change network or firewall configurations, or locking down staging environments to avoid last minute changes creeping in. What happens when you tell them that from now on software delivery is being automated all the way from source control to production, and that deployments will take place on a daily basis? How can they possibly keep up with these new challenges?

UNDERSTAND THE HUMAN NEEDS OF THE SECURITY TEAM

Start by understanding how they get their job done today; go through their workflow to really grasp the limitations and pressures they endure.

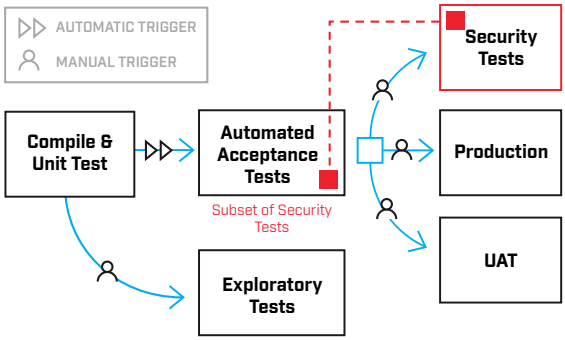
Next, explain how a deployment pipeline works and what controls are in place—such as ensuring functional adherence, no regressions, and any quality attributes you might be checking for (performance, reliability, etc). Explain how these controls are visible to everyone and how the pipeline stops when problems are found. Relevant security controls can be incorporated in a similar manner, actually reducing the security officer's workload and providing early insight on the software's compliance with security rules.

With a collaborative mindset, security officers, developers, and testers can come together and help each other with their respective gaps in knowledge and experience. Popular techniques that can be adopted for this purpose include pair programming (between a developer and a security person) or the "three amigos" conversation (focused on a feature's security concerns and testing approach).

INTRODUCE EARLY FEEDBACK FROM SECURITY TESTS

Moving from manual to automated processes has enabled operations (via infrastructure as code) to fit in the agile lifecycle. Implementations can take multiple forms, but the core benefit is to have everyone involved in delivery tuned to the same flow of delivery, receiving timely and relevant feedback.

Note that this doesn't imply the deployment pipeline needs to be fully automated, including security controls. In fact, at an early stage it might be advisable to keep manual security controls in the delivery pipeline. Security people get to keep familiar procedures and a sense of control, but there are immediate gains in terms of visibility of their work in the delivery process.



Shared visibility on security issues should be the holy grail for security officers. Security decisions no longer need to be made between a rock (cancel or rollback a release) and a hard place (release with recently found vulnerabilities). Instead security decisions can be made proactively instead of just piling up the security debt release after release.

In this example pipeline, we're following the recommendations from Jez Humble (co-author of the book Continuous Delivery) to keep deployment

pipelines "short and wide." We can run a subset of indicative security tests as part of the Automated Acceptance Test phase, leaving the bulk of the security tests to an optional phase later on.

We have found with clients in multiple industry sectors that taking a subset of indicative "weathervane" security tests and running these early in the deployment pipeline really helps to build confidence and trust in automated security testing; the weathervane tests show which way the longer suite is likely to go, yet we gain the ability to "stop the line" as soon as one of these early tests fails, leading to faster feedback.

USE LIGHTWEIGHT SECURITY TOOLS TO ENABLE GREATER FOCUS

In the last years a number of lightweight, command-line security tools have gained traction. We can distinguish between static analysis tools, security testing (dynamic analysis) tools, and security testing frameworks.

Static analysis tools have long been part of the developer's toolbelt for checking code quality. Oldtimer SonarQube and newcomer Code Climate are popular examples. SonarQube includes a number of language-specific security rules around protecting code from unwanted usage, hardcoded credentials, etc. Custom checks can be added or, alternatively, you can find language/framework specific security tools such as Brakeman for Ruby on Rails or Find Security Bugs for Java. These kinds of tools require little configuration to get started, but they can also generate many false positives initially—until you "groom" the rules.

Dynamic analysis tools typically launch a series of mechanic (but configurable) checks with varying degrees of complexity, from ports that shouldn't be open (for instance using Nmap) to SQL injection exploits (for example with sqlmap). They require a controlled environment where infrastructure, network, and application configuration is known (codified) and repeatable. These kinds of tools require more upfront effort to set up, but the results tend to be more accurate and cover a wide range of (attack) use cases. More importantly, results from both types of analysis can be fed into and visualized in the deployment pipeline fairly easily.

Other security checks and tools that can be incorporated in the pipeline include security scanning (e.g., Zapr or Arachni); inspecting build artifacts for viruses (e.g., ClamAV); or checking external dependencies for known vulnerabilities (e.g., bundler-audit for Ruby, NSP for Node, or SafeNuGet for .NET).

Finally, security testing frameworks provide a common way to specify and validate security scenarios. They abstract away the security tools being run, allowing higher level security discussions between developers, testers, security officers, and anyone else interested. Instead of debating results a posteriori (often requiring high cognitive effort for developers to remember their changes), a healthy pre-coding discussion takes place, uncovering potential weaknesses/attacks.

EXAMPLES OF SECURITY TESTS IN ACTION

Here we'll look at an example in action where security tests are run in a pipeline. In this particular case we're using sqlmap to check for SQL injection vulnerabilities. The pipeline is defined in GoCD, the Continuous Delivery tool from ThoughtWorks Studios: imgur.com/StV5Ecw

In the above linear example unit and acceptance tests pass but the pipeline stops when the security tests fail. If we drill down into the security test's job execution, we can see why it failed (sqlmap identified injection vulnerabilities): imgur.com/B9pLGNd

At this point we already have a centralized view of how any release candidate ranks in terms of the security controls that were baked in the deployment pipeline. No more compiling results from a number of security tools into a 50 page document that no developer will (voluntarily) read.

Let's look at another example, now using the gauntlt security framework (another example is BDD-Security) to specify the tests: imgur.com/MaWOJJT

The first scenario uses Nmap to check if a given port is closed as expected. The second scenario uses sqlmap to check for SQL injection vulnerabilities, as in the previous example.

A set of security/attack scenarios is defined and a color association (red/yellow/green) with the expected semantics: red means scenario failed (or attack succeeded if you will); yellow means it was not conclusive; and green means scenario passed (or attack failed).

FAVOR COMMUNICATION AND FEEDBACK OVER TOOL-DRIVEN PRACTICES

So now we not only have a centralized view of the security test results, but the results also follow a standardized format that's easy to understand by less technical stakeholders; we can clearly see which scenarios failed without having to understand how the underlying analysis tools work or how they report results.

Are we done once security controls are codified and integrated in the delivery pipeline? No, definitely not. What we have is a basis for trust, a set of validated assumptions about the application's security.

There are still many grey areas, unknown weaknesses that an attacker will be actively searching for. Security officers' expertise putting themselves in the role of an attacker, profiling threats, and poking the system for holes specific to the application's workflow are still required. They too need to be part of the delivery stream, either with manual gates in the deployment pipeline or scheduled at regular intervals, as long as the delivery flow is sustained.

The good news is that we moved the largely mechanical checks early in the pipeline, freeing up more time to explore those grey areas and avoiding time waste on subpar release candidates.

SUMMARY

To recap: automating security checks triggers conversations between developers, testers, and security folks. They get everyone on the same page, promote knowledge sharing, and establish common grounds and testable criteria for moving a candidate release through the deployment pipeline.

Throughout this article we've presented several practices and tools for bringing together and (re-)building trust between development and security. Each organization should adopt the ones that best fit their needs (and maturity level) to avoid regression to a blame culture. **Tools and practices alone don't lead to more secure systems; we also need frequent and open communication, feedback, and empathy for other people.**

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.

MANUEL PAIS is a DevOps advocate and InfoQ lead editor, with a mixed background as developer, build manager, and QA lead. Jack of all trades, master of continuous improvement, he believes validating ideas first, not software, is key to real progress. Specialist in Continuous Delivery by day. Specialist in Changing Diapers by night. Recently, Manuel happily joined the people-friendly technologists at Skelton Thatcher Consulting.

Executive Insights on Continuous Delivery

BY TOM SMITH

QUICK VIEW

01
DevOps/Continuous Delivery is in its infancy but its popularity is growing quickly as are the number of tools that facilitate speed and automation.

02
New companies that begin with a DevOps/Continuous Delivery methodology have a significant competitive advantage versus legacy companies that require a culture change to implement.

03
DevOps' evolution to No Ops should encourage all developers to understand operations, if not full-stack development, to ensure the longevity of their careers.

In order to gauge the state of the DevOps and Continuous Delivery movement in the "real world," we interviewed 24 executives actively involved in DevOps and Continuous Delivery. These are early-adopters and thought-leaders in the space.

Casey Kindiger, CEO, Avik Partners | **Ez Natarajan**, Vice President Cloud, Beyondsoft | **Tom Cabanski**, Director of Software Development, Blinds.com | **Kurt Collins**, Director of Technology Evangelism and Partnerships, Built.io | **Chris Madsen**, CEO, Circonus | **Yaniv Yehuda**, Co-Founder and CTO, DBmaestro | **Andreas Grabner**, Technology Strategist, Dynatrace | **Elaina Shekhter**, CMO, EPAM Systems | **Charles Kendrick**, CTO and Chief Architect, Isomorphic Software | **Baruch Sadogursky**, Developer Advocate, JFrog | **Topher Marie**, CTO, JumpCloud | **Edith Harbaugh**, CEO and Co-Founder, Launch Darkly | **Jessica Rusin**, Senior Director of Development, MobileDay | **Stevan Arychuk**, Strategic Marketing, New Relic | **Arvind Mehrotra**, President and Global Business Head, NIIT Technologies | **Zeev Avidan**, Vice President Product Management, OpenLegacy | **Richard Dominguez**, DevOps Engineer, Prep Sportswear | **Prashanth Chandrasekar**, General Manager of DevOps and Head of Operations, Rackspace | **Steven Hazel**, CTO, Sauce Labs | **Bob Brodie**, CTO, Sumo Heavy | **Dr. Chenxi Wang**, Chief Strategy Officer, Twistlock | **Scott Ferguson**, Vice President of Engineering, Vokal Interactive | **Adam Serediuk**, Director of Operations, xMatters

DevOps and Continuous Delivery is hot and growing. Its growth is outpacing available talent. As such, companies are looking for developers with some exposure to operations and operations professionals that relate well to developers. Ultimately a successful DevOps professional has to be good with people, it's a collaborative role that can be very political in established enterprises.

Here's what we learned from the conversations:

01

The time respondents have been involved with DevOps varies greatly with the majority of respondents claiming to have been **involved for more than five years**, or throughout their career. Additionally, newer firms were more likely to have started with a DevOps methodology. It's much easier to start a company with a DevOps culture than it is to change the culture of an enterprise with a traditional IT infrastructure.

02

Collaboration and integration are the key features of DevOps with the benefits being able to build and deploy faster, streamlining delivery and maintaining a highly integrated feel and team. Ultimately this is a function of team members removing arbitrary boundaries through communication and defined processes promoting collaboration in a cooperative fashion. Also, DevOps provides a continuous feedback loop. It's important to share the feedback with all relevant team members as quickly as possible to collaborate on problem solution, and opportunities.

03

Code being delivered with great frequency, from hourly to every two weeks, is the feature of continuous delivery and continuous integration which are mentioned synonymously. This enables developers to get a lot of small changes out to customers with great frequency thereby being more responsive to their needs. The process is highly reliant on automated testing and deployment and results in greater responsiveness with fewer errors and less risk.

04

The most important elements of DevOps and Continuous

Delivery are the **streamlined deployment process**, the simplicity and repeatability of the process, thanks to automation, and the culture change that affects the organization as a whole. DevOps starts tech centric and then permeates the entire organization with everyone in the company becoming responsive to customer feedback.

05

The greatest value being seen by the implementation of DevOps and Continuous Delivery is the **speed to market** with a stable and reliable product. This reduces the cost of getting a product to market while increasing the responsiveness to the customer. It enables you to improve as quickly as possible before the competition does. Developers are more productive and happier since they're able to get virtually real-time feedback. As you begin to automate you increase the predictability of the product and deployment times.

06

The skills that make someone good at DevOps and Continuous Delivery are first and foremost **the ability to get along, and communicate, with others** given the collaborative nature of the DevOps methodology. Next is having a thirst to learn new things and being willing to change and be flexible. Full-stack development skills and a development mindset are beneficial as well. Nonetheless, you need to be focused on the customer and on the organization rather than the individual. As one respondent said, "DevOps is a mindset rather than a skillset."

07

Tools are the most frequently mentioned when asked how DevOps have evolved since its inception; however, with the proliferation of tools, there's is still significant concern with their level of sophistication and ability to meet the increasing demands of users. Tools that support DevOps remain relatively rudimentary, and most organizations that are effective with DevOps use a lot of homemade automation.

08

Politics and the existing culture are by far the greatest impediments to the success of DevOps initiatives at a company. "Old mindsets," "resistance to change" and "breaking departmental boundaries" are all reasons given for these challenges. A lot of companies do not understand, let alone embrace, the culture shift that's necessary to successfully implement DevOps. Humans are inherently afraid of change, particularly if the status quo is deemed "safe" or "satisfactory." Entrenched methodologies – even if they don't demonstrate much success – are comfortable for managers and employees. The ability to experiment – and to fail and learn – is crucial to success implementing a DevOps methodology.

09

Executives have a couple of concerns regarding DevOps and Continuous Delivery—the tendency for people to embrace, and chase, **silver bullets and the ongoing complexity** of the movement which can be overwhelming if a team, and a company, are not prepared to deal with the evolution and the new tools that are coming to market on a daily basis.

10

The greatest opportunities for DevOps and Continuous Development are the continued evolution of **tools and automation**. Some respondents expect to see container technology from Microsoft and other big players. Others see real-time feedback and more structure so that DevOps evolves to No Ops—the way Netflix is structured. Automation will enable multiple levels of testing during the development process so anyone will be able to push a button to initiate a release of a thoroughly tested product.

11

For developers who want to become involved in DevOps and Continuous Delivery, they need to focus on developing **end-to-end solutions**. Produce high-quality working code that is well tested, works reliably and performs well in production. Go from 95% code coverage to 99.9999%. Recognize and include the operations aspects of software into development. Build for five-minute release patterns—microservices. Get your head out of your computer and talk to others, understand what happens with your code and how what you are doing adds value to your company and meets end-users' needs.

12

One question raised by several executives at the conclusion of the interview seems to be "the elephant in the room"—**has anyone really proved the ROI** of DevOps and Continuous Delivery? Intuitively it makes sense that speed to market, reduction of errors, testing and being responsive to customers in a timely manner results in lower costs and more revenue. However, someone needs to develop a true business case using data to prove the ROI of DevOps and Continuous Delivery.

Another point was raised by several of the respondents. DevOps is new, it's big and it's somewhat amorphous. We talk about DevOps being "all or nothing;" however, it's really a continuum where you can continue to reduce to time market and get smarter. What's most important is building what's right for your organization so you can continuously improve while you're continuously developing, integrating and deploying.

The executives we spoke with are proponents of the DevOps and Continuous Delivery movement; however, they want to see it succeed—success breeds success. We're interested in hearing from developers, and other IT professionals, to see if these insights offer real value. Is it helpful to hear others' perspectives from a more senior perspective. Are their experiences and perspectives consistent with yours? We welcome your feedback at research@dzone.com.

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.

The Critical Role of a Universal Artifact Repository Manager in a CD Ecosystem

Companies in every industry are sharing the pain of moving to a modular software architecture in order to deliver software faster. Automation is the only proven way to achieve rapid software delivery cycles, and a universal artifact repository manager plays a critical role in automating end-to-end software delivery to consumers and devices.

UNIVERSAL

Today's development ecosystems are constructed of many binary technologies and include everything from Docker images to Node.js packages to RubyGems, Maven, Python, NuGet and more. An artifact repository manager that is universal automates delivery of software for all development teams, regardless of the technology they are using.

HIGHLY AVAILABLE

The CD pipeline is a mission-critical element of the software delivery workflow, and companies cannot tolerate any downtime that halts automatic processes. A universal artifact repository manager that offers high availability ensures the CD ecosystem is up and running at all times.

SECURITY AND ACCESS CONTROL

A universal artifact repository manager offers many layers of security and authentication to ensure that software artifacts and builds are protected both from malicious intervention, as well as from innocent accidents. This is the kind of strict access control that any CD ecosystem needs to ensure that the delivery pipeline is not breached.

CONTINUOUS DISTRIBUTION

Integrating with JFrog's distribution platform changes the "D" of CD from Delivery to Distribution. Bintray.com lets a universal artifact repository manager automate the continuous distribution of your software out to users and devices over rapid CDN.

While you can implement continuous delivery in different ways, a universal artifact repository manager at the core of your CD ecosystem will accelerate your software delivery by allowing full automation of your development and build process all the way to distribution.

WRITTEN BY YOAV LANDMAN
FOUNDER AND CTO AT JFROG LTD

Artifactory BY JFROG

JFrog Artifactory shortens release cycles by automating any CD ecosystem through support for all major packaging formats, build tools, and CI/CD servers.

CATEGORY
Universal Artifact Repository

NEW RELEASES
Two weeks

OPEN SOURCE?
Yes

CASE STUDY

Technology giant Oracle employs 40,000 development staff globally. The company needed an artifact repository that supported the massive scale of their operations, which included millions of artifacts stored in terabytes of data in several global data centers. Oracle selected Artifactory because it was the only solution able to support such a massive scale while maintaining performance and reliable access to artifacts. Today, much of the company's development ecosystem is built around Artifactory, which became a development hub. The main reasons quoted for Artifactory's success at Oracle were its support for enormous scale, its universal support for packaging formats, build tools and CI/CD servers, and prompt and attentive technical support.

STRENGTHS

- Universal: Supports all major packaging formats, build tools, and CI servers.
- Enterprise-ready: High availability, dynamic and unlimited storage, multi-push replication.
- Security: Multiple authentication protocols and fine-grained access control to artifacts.
- Mission Control: Single access point to manage and monitor all global Artifactory installations.
- Docker: Full support for Docker providing multiple secure, private Docker registries.

NOTABLE CUSTOMERS

- Amazon
- MasterCard
- Oracle
- Google
- Netflix
- VMware
- LinkedIn
- Tesla

BLOG jfrog.com/blog | **TWITTER** @jfrog | **WEBSITE** www.jfrog.com

SOLUTIONS DIRECTORY

This directory contains software configuration, build, container, repository, and application performance management tools, as well as many other tools to help you on your journey toward Continuous Delivery. It provides free trial 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.

PRODUCT	CATEGORY	FREE TRIAL	WEBSITE
AccuRev by Micro Focus	Software Configuration Management	No	borland.com
Agile Software Factory by Grid Dynamics	Application Release Automation	No	griddynamics.com
Amazon ECS	Container Management	No	aws.amazon.com
Ansible by Red Hat	Configuration Management, Application Release Automation	Yes	ansible.com
Apache Ant	Build Management	Yes	ant.apache.org
Apache Archiva	Repository Management	Yes	archiva.apache.org
Apache Maven	Build Management	Yes	maven.apache.org
Apache Subversion	Software Configuration Management	Yes	subversion.apache.org
AppDynamics	Application Performance Management	No	appdynamics.com
Appium	Automated Web and Mobile Testing	Yes	appium.io
Artifactory by	Repository Management	No	jfrog.com
Bamboo by Atlassian	Continuous Integration, Application Release Automation	No	atlassian.com
BMC Release Lifecycle Management	Application Release Automation	No	bmc.com
Buildbot	Continuous Integration	Yes	buildbot.net
Buildmaster by Inedo	Application Release Automation	No	inedo.com
CA Release Automation	Application Release Automation	No	ca.com
Chef	Configuration Management	Yes	chef.io
Chef Delivery	Application and Infrastructure Release Automation	No	chef.com
CircleCI	Continuous Integration	No	circleci.com
Concurrent Versions Systems (CVS)	Software Configuration Management	Yes	savannah.nongnu.org/projects/cvs
Cucumber	Automated Rails Testing	Yes	cucumber.io
Datadog	IT Stack Performance Management	No	datadoghq.com

PRODUCT	CATEGORY	FREE TRIAL	WEBSITE
DCHQ	Container Management	Yes	dchq.io
DLM Automation Suite by Redgate Software	Database CI, Release Automation, Database Lifecycle Management	No	red-gate.com
Docker Swarm	Container Management	Yes	docker.com
ElasticBox	Container Management	No	elasticbox.com
ElectricFlow by Electric Cloud	Application Release Automation	Yes	electric-cloud.com
FitNesse	Acceptance Testing Framework	Yes	fitnesse.org
Git	Software Configuration Management	Yes	git-scm.com
Go by Thoughtworks	Application Release Automation	Yes	go.cd
Gradle	Build Automation	Yes	gradle.org
Gridlastic	Automated Web Testing	No	gridlastic.com
Helix by Perforce	Software Configuration Management	No	perforce.com
Hudson by Eclipse	Continuous Integration	Yes	hudson-ci.org
IBM Rational Strategy	Software Configuration Management	No	ibm.com
Incident.MOOG	Event Management	No	moogsoft.com
Jenkins	Continuous Integration	Yes	jenkins-ci.org
Jenkins Enterprise by CloudBees	Continuous Integration, Application Release Automation	No	cloudbees.com
JMeter	Web and Java Testing	Yes	jmeter.apache.org
JUnit	Unit Testing Framework	Yes	junit.org
Kubernetes by Google	Container Management	Yes	kubernetes.io
Logentries	Log Management and Analytics	No	logentries.com
Mercurial	Software Configuration Management	Yes	mercurial-scm.org
Nagios Core	Infrastructure Monitoring	Yes	nagios.org
New Relic	Application Performance Management	No	newrelic.com
Nexus by Sonatype	Repository Management	Yes	sonatype.org
NuGet	Repository Management	Yes	nuget.org
NUnit	Unit Testing Framework	Yes	nunit.org
ONE Automation by Automic	Application Release Automation	No	automic.com
OpsGenie	Monitoring Alert Software	No	opsgenie.com
PagerDuty	Monitoring Alert Software	No	pagerduty.com
ParallelCI by Codeship	Continuous Integration	No	codeship.com
Parasoft	Automated Web and API Testing	No	parasoft.com

PRODUCT	CATEGORY	FREE TRIAL	WEBSITE
Plutora	Application Release Automation	No	plutora.com
Puppet by Puppet Labs	Configuration Management	No	puppetlabs.com
Rake	Build Automation	Yes	github.com/ruby/rake
Ranorex	Automated Web and Desktop Testing	No	ranorex.com
RapidDeploy by MidVision	Application Release Automation	No	midvision.com
Rapise by Inflectra	Automated Web Testing	No	inflectra.com
RepliWeb for ARA by Attunity	Application Release Automation	No	attunity.com
Rocket ALM by Rocket Software	Application Release Automation	No	rocketsoftware.com
Sahi	Automated Web Testing	No	sahipro.com
Salt by SaltStack	Configuration Management	Yes	saltstack.com
Sauce Labs	Automated Web and Mobile Testing	Yes	saucelabs.com
Selenium WebDriver	Automated Web Testing	Yes	seleniumhq.org
Serena Deployment Automation	Application Release Automation	No	serena.com
Shippable	Continuous Integration	No	shippable.com
SnapCI	Continuous Integration	No	snap-ci.com
SoapUI by SmartBear Software	Automated Web and API Testing	Yes	soapui.org
Solano Labs	Continuous Integration	No	solanolabs.com
Team Foundation Server by Microsoft	Software Configuration Management	No	microsoft.com
TeamCity by JetBrains	Continuous Integration, Application Release Automation	No	jetbrains.com
Tellurium	Automated Web Testing	No	te52.com
TestingBot	Automated Web Testing	No	testingbot.com
TestNG	Unit Testing Framework	Yes	testng.org
TravisCI	Continuous Integration	Yes	travis-ci.org
Urbancode Build by IBM	Continuous Integration, Build Management	No	ibm.com
Urbancode Deploy by IBM	Application Release Automation	No	developer.ibm.com
Vagrant by Hashicorp	Configuration Management	Yes	vagrantup.com
VictorOps	Monitoring Alert Software	No	victorops.com
Watir	Automated Web Testing	Yes	watir.com
Windmill	Automated Web Testing	Yes	getwindmill.com
XL Deploy by Xebia Labs	Application Release Automation	No	xebialabs.com
xUnit	Unit Testing Framework	Yes	xunit.github.io

Introduction to CI in a Docker World

Continuous Integration (CI) and Continuous Delivery (CD) practices have emerged as the standard for modern software testing and delivery. Docker-based solutions, such as Shippable, accelerate and optimize CI/CD pipelines, making it easy for developers and operations teams to collaborate. Best of all, executing end-to-end CI/CD pipelines is fully achievable in the cloud with Docker, Shippable, and Git-based code repositories, without investing in or managing your own infrastructure.

A typical "containerized" CI/CD pipeline consists of three cloud-based solutions, one for source control (like GitHub or Bitbucket), one for container images (like Docker Hub, Amazon ECR, or Google GCR), and one for managing testing and flow through the pipeline, like Shippable. The pipeline will be kicked off by a source code commit to a code repository. The commit will trigger Shippable CI/CD to spin up a build job inside a container, and, upon successful completion of that job, push an image up to a container registry such as Docker Hub. From there, the image will be deployed automatically into Shippable for functional, integration, and smoke testing.

Integrating containers into CI/CD pipelines has helped many organizations accelerate system provisioning, reduce job time, increase the volume of jobs run, enable flexibility in language stacks and improve overall infrastructure utilization. The immutability of container images ensures a repeatable deployment of what's developed on the developer's machine, through testing (unit and staging) tested through CI and ultimately in production.

For example, Docker Hub is an online registry service for storing and retrieving Docker images based on Docker, an open platform to build, ship and run distributed applications, anywhere. At the core of the Docker solution is a registry service to manage images and the Docker Engine to build, ship and run application containers. Google and Amazon have registries for their respective systems. Bitbucket and GitHub are Git-based online services for creating and contributing to public and private software projects using a powerful collaborative development workflow.

For more information and to start building pipelines with Docker, you can visit http://content.shippable.com/dzone to learn more and set up your first containerized app pipeline.

WRITTEN BY TOM TRAHAN
VICE PRESIDENT, SHIPPABLE

Shippable CI/CD BY SHIPPABLE

shippable

Ship code faster with the Shippable CI continuous integration platform. Hybrid CI for on-premise, cloud, and container-based builds.

CATEGORY
CI/CD Tools

NEW RELEASES
Shippable 3.0

OPEN SOURCE?
No

CASE STUDY

"If your goal is to commit code often, test it, and ship to production as fast as possible to keep up with your company's changing features and initiatives, there is no better product than Shippable. Using Shippable cut down our testing time from over 20 minutes to 8 minutes."

—CTO AT MOJOPAGES

STRENGTHS

- Shippable CI is the only CI platform that gives you the convenience of a SaaS with the integrations and flexibility of running your own solution.
- Support for GitHub, GitHub Enterprise, Bitbucket, Bitbucket Server, and GitLab
- CI that lives where you work—in your source code repos
- The only hybrid CI platform available—run your builds anywhere, even behind your firewall
- Whether on your infrastructure or ours, we manage and support your CI platform for you
- Complete flexibility to dedicate the right resources for your build

NOTABLE CUSTOMERS

- Cisco
- Engagespark
- SAP
- Concur
- Lithium
- Google
- Pushspring

BLOG blog.shippable.com | **TWITTER** @beshippable | **WEBSITE** www.shippable.com

glossary

AGILE SOFTWARE DEVELOPMENT
A software development methodology and philosophy, focused on user feedback, software quality, and the ability to respond quickly to changes and new product requirements.

APPLICATION RELEASE AUTOMATION (ARA) A practice of deploying software releases to various environments and their configurations with as little human interaction as possible.

BEHAVIOR-DRIVEN DEVELOPMENT (BDD)
An evolution of test-driven development that focuses on collaboration between development and business stakeholders to define user stories that determine the development of the application using a human-readable DSL.

BUILD AGENT A type of agent used in continuous integration that can be installed locally or remotely in relation to the continuous integration server. It sends and receives messages about handling software builds.

BUILD ARTIFACT REPOSITORY Centralized storage for all binaries used during build. Simplifies dependency management and build processes, helps maintain security and consistency across teams, helps make automated deployment practical and scalable.

CAPACITY TEST A test that is used to determine the maximum number of users a computer, server, or application can support just before failing.

CONFIGURATION DRIFT How software and hardware configurations become inconsistent with the master version due to manual and ad hoc changes (like hotfixes) that are not committed back to version control. Often a significant source of technical debt.

CONFIGURATION MANAGEMENT A process for establishing and maintaining consistent settings of a system. These solutions also include SysAdmin tools for IT infrastructure automation (e.g. Chef, Puppet, etc.).

CONTAINERIZATION 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 (CD) A software production process where the software can be released to production at any time with as much automation as possible for each step.

CONTINUOUS DEPLOYMENT A software production process where changes are automatically shipped to production without any manual intervention.

CONTINUOUS INTEGRATION (CI) 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.

DEPLOYMENT A term that refers to the grouping of every activity that makes a program available for use and moving that program to the target environment.

DEPLOYMENT PIPELINE A process for getting software from version control to the production environment by moving builds of that software through multiple stages of testing and production.

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.

EXPLORATORY TESTING A manual testing strategy where human testers have the freedom to test areas where they suspect issues could arise that automated testing won't catch.

INTEGRATION TESTING Testing that occurs after unit testing, but before validation testing, where individual software components are combined and tested as a single group to ensure they work as a whole.

ISSUE TRACKING A process that allows programmers and quality assurance personnel to track the flow of defects and new features from identification to resolution.

LEAD TIME The time it takes to move work in progress (WIP) to a finished state in a manufacturing plant. In software development, this is represented by moving code changes to production.

MEAN TIME BETWEEN FAILURES (MTBF)
Used to measure reliability of a system or component, calculated by averaging the time between system failures.

MEAN TIME TO RECOVERY (MTTR) The average time it takes a system or component to recover from a failure and return to production status.

PLATFORM-AS-A-SERVICE (PAAS)
Provides languages, libraries, services, and tools that allow developers to build and deploy applications in the cloud without worrying about underlying OS-level infrastructure (or below).

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

ROLLBACK An automatic or manual operation that restores a database or program to a previously defined state.

SELF-SERVICE DEPLOYMENT The action of automating deployment processes enough for developers to allow project managers or even clients to directly control deployments.

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.

TECHNICAL DEBT A collection of consequences—such as slow deployment times or performance—that will eventually result from a system or application that is not properly maintained or reviewed for quality and functionality.

UNIT TESTING A testing strategy in which the smallest unit of testable code is isolated from the rest of the software and tested to determine if it functions properly.

USER ACCEPTANCE TEST The final phase of software testing where clients and end-users determine whether the program will work for the end-user in real world scenarios. This stage is also known as beta testing.

VIRTUAL MACHINE (VM) A software emulation of a physical computing resource that can be modified independent of the hardware attributes.