The Practical Guide to Enterprise DevOps and Continuous Delivery

MICRO FOCUS®

Contents

About the Author–

Julian Fish is the Director of Products for Micro Focus' release and deployment management solutions. Julian has over 15 years' experience in the IT industry, starting in Quality Assurance before moving into domains as diverse as infrastructure management, database administration, software development, release engineering, and IT operations. In recent years, Julian has helped many large organizations transform their development and release management processes. Julian holds a degree in Applied Geology with honors from the University of Leicester.

Introduction

It is very likely that you found this book and chose to read it to gain a better understanding of the adoption of Continuous Delivery (CD) and DevOps in the enterprise software space. You and your organization have concerns about the increasing rate of change in the market where you compete and are looking at ways to drive up the quantity and speed of change to meet time-to-market pressures. The IT landscape has become more service centric, with customer demand for improved services at an all-time high. In virtually every business today, the interaction between consumer and organization has become increasingly digital. Consumers are accustomed to continuous change and improvement in the apps they use, and you have to be able to offer that to your customers.

This book is for enterprise thought leaders looking to increase the velocity and volume of valuable change to their critical and revenue-generating IT services. Achieving this while remaining mindful of preserving the stability, security, and availability of the services they deliver is difficult. This book will show how, as companies look for ways to innovate faster to keep pace with more agile competitors or to take the lead in a crowded marketplace, they are exploring many ways to accelerate the safe delivery of strategic business solutions. These companies have set their goals beyond simply speeding up the delivery of releases. They strive to change their very corporate culture to one where applications evolve continuously, leaving behind forever the notion of sequential, serial releases with long lead times and small feature delivery.

If your intention is to adopt a Continuous Delivery strategy for enterprise applications as part of your DevOps transformation program, read on – this book is for you.

"In the past, business success was all about size: The large eat the small. Today, business success is all about speed: The fast eat the slow."

Daniel Burrus

Futurist

What is Continuous Delivery?

Continuous Delivery is a software engineering approach that affects the entire process of releasing software. Automation of processes in the development, testing, and deployment of applications is a vital component of CD, but a true CD solution is not something that you can simply take out of a box. It may require a degree of organizational change, and everyone who touches the enterprise software release process must adopt it, not only development but also QA, operations, and even the business as a whole.

Over the last two decades, the adoption of agile development methodologies has had a dramatic effect upon the way enterprises deliver software. Development teams can now create and update applications much more rapidly, and they view what they produce as a continuously evolving, continuously improving code base, tightly aligned with the needs of the business. Until recently, however, the teams downstream have not always embraced this newfound agility equally.

Continuous Delivery is a mechanism designed to increase both agility and velocity through the deployment process. Its ultimate success depends on breaking down the dividing lines between development, QA, and operations, so that the entire software lifecycle runs in a continuous and business-aligned fashion.

What is DevOps?

There is a huge amount of contradictory information around the term DevOps. All manner of organizations view DevOps as a part of the "latest IT fashion" and use it to promote their book/software/hardware/services/seminars. It is easy to slap a DevOps logo on your product, but that does not make it DevOps.

Wikipedia describes DevOps as a "culture, movement or practice that emphasizes the collaboration and communication of both software developers and other information-technology (IT) professionals whilst supporting the practices of automating software delivery and infrastructure changes. It aims at establishing a culture and environment where building, testing and releasing software can happen rapidly, frequently and more reliably."[1] It is also a combination of development and operations practices, policies, procedures, and process. In the enterprise software world, DevOps provides a vehicle for organizational transformation from siloed, traditionally adversarial groups to collaborative, shared ownership teams with a common goal and collective responsibility.

[1]Taken from https://en.wikipedia.org/wiki/DevOps

The previous generation of enterprise transformation projects occurred with the rapid adoption of agile practices in the 2000s. Often, today's DevOps initiatives begin when an executive sponsor hears the stories of successful implementations in high-value, low-footprint organizations. When Netflix, Facebook, and Google successfully implement DevOps to get a new capability to market more quickly, they ask why we can not do the same. DevOps defines practices, processes, and strategies to increase collaboration, efficiency, and communication between teams involved in the end-to-end Dev to Ops landscape. Adjacent teams will be affected and brought into the initiative, and they include (but are not limited to) architecture, infrastructure, QA, production support, and change and release teams. The intention is to consolidate and align these teams, in terms of both process and technology, to allow organizations to streamline their end-to-end application delivery process. This, in turn, will lead to faster, higher-quality releases with reduced risk and lower costs.

This book approaches DevOps from a software/application engineering and deployment perspective, but there are a number of other key disciplines that need to be part of any major DevOps initiative. These include:

- **Code** – Code development and review, static code analysis, continuous integration tools
- **Build** – Version control tools, code merging, build status
- **Test** – Continuous testing, test automation and results to determine performance
- **Package** – Artifact repository, application pre-deployment staging
- **Release** – Change management, release approvals, release automation, provisioning
- **Configure** – Infrastructure configuration and management, infrastructure as code tools
- **Monitor** – Applications performance monitoring, end user experience

The main areas of focus of this book are code, build, test, package, and release, with the areas around configuration and monitoring falling beyond its scope.

What this book will provide

Many enterprises need help transforming their organization to support Continuous Delivery – not simply through a software vendor but via a delivery partner. This partner needs to gain a deep understanding of the enterprise, its business initiatives, who is leading these initiatives, and what help is required.

Adopting true DevOps and Continuous Delivery practices is a difficult and complex undertaking for the typical large enterprise. This book makes life easier by giving you access to Micro Focus' extensive experience in some of the world's most advanced IT organizations. Our strategic view enables you to learn what modern organizations need to do to make the adoption of DevOps work, to sustain its effectiveness, and to ensure that it is continually improving.

So, let's start at the beginning.

1

The Business Problem

Markets in which enterprises compete are changing faster than ever, because of the increasingly technological nature of products and services. Even the most mundane products are digital or are marketed through digital channels. Rapid change is driven by the expectations of today's digital-savvy customers and the race for disruptive innovation to outwit competitors. Globalization and the minimal cost of entry into a market magnify the effect of competition as new players enter, disrupt and destroy markets built over centuries in weeks and sometimes days with novel business practices and globalized, digital-based supply chains.

Accelerating change in your business to meet these threats and exploit these opportunities means shifting to new marketing paradigms like social and mobile and changing to dynamic supply chains; contingent workforces; and small, specialist, independent suppliers (who themselves might be disrupted out of existence tomorrow).

But most of all, it is the digital transformation of your software development and delivery process that will give you the edge to win in the market.

The nature of software itself has changed over the past decade with the proliferation of Web, mobile and now Internet of Things (IoT) applications. There are 1.5 million apps in the Apple App Store (opened in June 2008) and 1.8 million in the Google Play Store (opened in December 2009). Today, every market sector and vertical segment understands the value that apps can bring to organizations. The switch to app-based delivery of value to the customer has seen software increasingly componentized and architected with service-oriented models in mind. Many legacy applications are already re-engineered into libraries of standardized widgets and components. Achieving the technical advantages of re-architecting applications to allow faster, more feature-based delivery is unfortunately very expensive and takes a long time. Projects fail frequently, are often short lived, and rarely deliver any differentiating value. Take solace from the fact that your competitors probably have equivalent initiatives underway and are experiencing the same outcomes that you are.

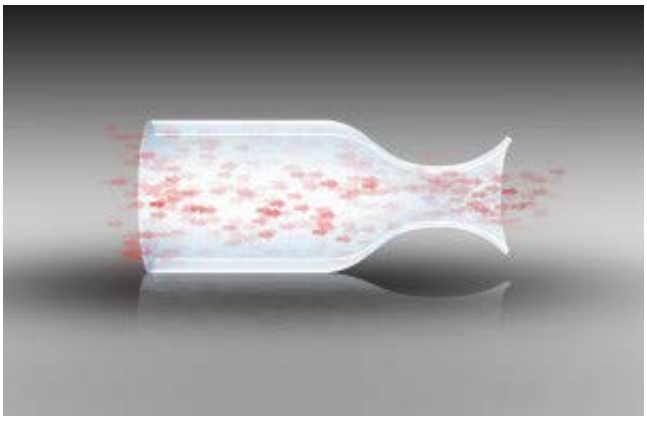
The adoption of agile practices for application and software project management has allowed the business to request finer and finer (incremental) levels of change to existing applications. Development teams have set the expectation that they can develop in an incremental, rapid, "just in time" style of delivery when changes are requested. This approach has driven development costs down, increased quality, and reduced the business risk inherent in any change to a stable system. However, the volume of change this creates has overwhelmed the downstream process of releasing the software into production. And this has also led to an orders-of-magnitude increase in cross-component and cross-application dependency, which is now so much more intricate that it directly and adversely affects the delivery of proposed changes.

Your ability to deliver features that the market wants rapidly and safely is your most important commercial differentiator, not the speed at which you can make the changes to the code.

The Shared Objective: Move fast without breaking things, but if you do break something, fix it fast

If rapid development is not differentiating, then what is? Proponents of CD argue that it is rapid **deployment** – the actual delivery of the software into production, quickly and free of errors, that delivers value to the business. What use is the latest and greatest new feature in your software if it is sitting in a QA lab and not on a customer-facing device?

The key objective for any industry, be it engineering, manufacturing or software delivery, is to shorten the time to market. This means the delivery of finished software to the end users, who in turn use it to provide new or updated services to customers or provide rapid feedback on the product. Who would not prefer to find out that the latest innovation from your Product Management organization fails to address customer requirements three months into its lifecycle as opposed to twelve months later? The principles and practices of Continuous Delivery allow organizations to address the inefficiencies between the business, development, QA and operations, reducing cycle time and eliminating wasteful practices from the end-to-end process. Applying Lean[2] principles and cycles such as those proposed by Deming, many organizations have found that increased rates of delivery and early identification of issues lead to reductions in cost and increases in quality. **Every software executive wants to improve delivery rates, quantities, and accuracy without reducing quality or increasing costs.**



At one national pharmacy chain ...

The core application for filling prescriptions traditionally had been updated once a year. However, the dynamics of this highly competitive market forced the company to update more frequently to keep up with competitive initiatives. It undertook a DevOps initiative specifically to enable three major releases each year.

[2] https://www.lean.org/WhatsLean

One major challenge to the needed organizational changes lies in how the teams are measured. Historically, Ops has been scored based on service-level agreements (SLAs), specifically measures that rate system uptime, incident volume, and time to restore services, but not on the number of changes delivered to production. Development is rebuked for late and incomplete code but not measured on the quality or production stability of applications. The IT Service Management (ITSM) practices that have been encouraged and significantly adopted by many organizations are often seen to be in direct conflict with the agile mantras of delivering working software in a continuous manner. According to the agilemanifesto.org's Principles Behind the Agile Manifesto, "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software." This is not actually the case, but ITSM administrators take seriously the 100% uptime objective they have had, and they know that changes are what tarnish that perfect score.

How, then, to compete with a startup that is disrupting your market with incremental capabilities using two-week release cycles and getting immediate customer feedback?

Start with Continuous Delivery.

Continuous Delivery: Deployment Retooled for High-Velocity Competitive Change

2

The adoption of Continuous Delivery (CD) is a journey, and it is one that should not be undertaken lightly. The organizational impact and political implications of CD practices are almost as significant as the benefits that they deliver. Having this knowledge from the outset allows individuals to set appropriate expectations and maintain focus on a well-defined objective.

Jez Humble, a Silicon Valley technologist and consultant, is widely credited with solidifying the goals and principles for CD in his 2010 book, ***Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*** (Humble, Farley 2010). The principal objective of Continuous Delivery, Humble writes, is to deliver high-quality, valuable software in an efficient, fast, and reliable manner. This can be traced back to the objectives of the Agile Manifesto published in 2000, which stated: "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software."

Three kinds of processes exist within organizations: documented processes, undocumented processes, and day-to-day working practices. Many organizations rely upon "tribal knowledge" both in App Dev and IT Ops. Key individuals act as heroes because they hold all the knowledge in their heads and are the only people able to rescue a release that is failing. When every application depends on a "hero," it is common to have a release window that needs dozens, if not hundreds, of participants to get code into production successfully.

There is a more efficient and practical route to production code delivery.

Allianz: DevOps and Resourcing

Allianz faced challenges with the resource costs and time to market of their main business application. The AMOS (shared service) team was deploying up to three releases a day for this application, which required both Oracle database and WebSphere deployments. Allianz implemented a "single click" deployment solution to remove the need for highly paid technical resources to be involved in these deployments. Using Deployment Automation, Allianz repurposed two highly skilled resources per release, reduced release time, and increased delivery of business change.

Click here to read the full Allianz Case Study.

Jez Humble lists the following seven core principles for ensuring effective DevOps adoption:

- Create a repeatable, reliable process
- Automate almost everything
- Keep everything in version control (even the automation scripts)
- If it hurts, do it more frequently
- Build quality in
- "Done" means released
- Everyone is responsible for the delivery process

We will examine these principles more closely as a way of addressing the CD challenge. (In Chapter 3, we will examine where to begin in Continuous Delivery adoption, and in Chapter 4, we will take a deeper dive into deployment pipeline automation.)

Create a repeatable, reliable process

Your development, QA, and deployment teams have probably articulated the goal of repeatable process already – each within its own silo – resulting in three different processes and mechanisms to get code into the correct environments. DevOps teams focus on extending the agile process from development into testing and deployment. Effective adoption of Continuous Delivery should result in a process continuum that brings together these teams and blends them into one continuous agile process. Controls and compliance principles needed by Ops and the business are retained (and automated where possible), and well-established practices and working processes are preserved where appropriate. It is all about streamlining the processes and removing the process boundaries.

Automate as much as possible

Getting code from development to test to User Acceptance Testing (UAT) to production need not be difficult or complex. However, as the adage states, "to err is human," and this is more common during the release or deployment of applications to production than it should be. Customers frequently comment that most production outages are not due to errors in deployment or configuration but are due to poor code or application quality. A release engineer who must perform 200 steps listed in a spreadsheet late on a Friday night will not execute every one of them correctly every time. Most errors and inefficiencies that delay deployments are found to be human errors.

Humble's advice, backed up by Gartner and other key analysts is this: *If you do nothing else, start automating your deployments.* It does not matter whether you start in development or in operations – automate at one end of the deployment pipeline and then work your way towards the other end. Automation simplifies and accelerates development, testing, and deployment, and it eliminates most common errors.

The reality is that when Continuous Delivery or DevOps transformation projects are put into practice, automation most commonly begins in Dev, with the implementation of Continuous Integration (CI). CI is the practice of building the application every time a developer commits a change back into the source code repository. Usually this is achieved with an automated build and unit testing capability. What this means is that, if the developer breaks the build or if new code fails the battery of automated tests, that feedback is delivered immediately so that the repair or error can be remedied before the broken code affects other developers or the test teams.

The immediate identification of issues and of who broke the build radically alters the development paradigm. Where we used to build once a day and then spend hours trying to unpick 50 or more code commits to find the culprit, we now know instantaneously – and so does the developer. Many organizations struggle to have a single, common build process across their teams, as more mature teams use well-established CI practices while other teams do not even have a common process for build. Standardizing build processes and then implementing CI fulfills the most basic development requirement for Continuous Delivery and can be seen as the essential first building block upon which to base CD implementations. Many CI build tools even extend their capabilities towards extended automation and pipeline management. The challenge of how to adopt the components of CD across QA and operations is one that many enterprises are now trying to address, and a number of technological solutions have been tried with varying degrees of success.

Version control for all artifacts.

"Version control provides a single source of truth for all changes. That means when a change fails, it's easy to pinpoint the cause of failure and roll back to the last good state, reducing the time to recover. Version control also promotes greater collaboration between teams. The benefits of version control shouldn't be limited to application code; in fact, our analysis shows that organizations using version control for both system and application configurations have higher IT performance."

2014 State of DevOps Report (Puppet Labs)

Keep everything in version control

Developers should commit their code changes to a common repository that versions every change. The more development teams your organization has, the more repositories there will be. Repository sprawl is becoming a significant overhead and risk as enterprise-sized organizations struggle to put controls in place and to maintain oversight and governance of the source code. Development tools, once under the control of a central IT function, are now housed on a developer's server under the desk. Under the guise of agility, development teams have taken control of their repository and, in doing so, have disconnected themselves from the Dev collective and the security of corporate IT. Development leaders have lost control of projects, lack insight into the status of tasks, and have left their valuable intellectual property to roam freely and insecurely, somewhere on the WAN.

In order to transform simple version control to a more continuous build model, all changes in motion are merged back to the trunk on at least a daily basis and are subjected to a common build. The output of that build is an executable, which is also versioned into the repository. True DevOps adoption requires commitment to versioning of all project artifacts from one end of the application lifecycle to the other – e.g. requirements, test scripts, release scripts, UI specifications, prototypes, build, baselines, and so on.

If it hurts, do it more frequently

IT people are human; they tend to back away from process steps that are difficult, prone to error, or likely to expose them to criticism. The DevOps ethic is to confront these challenging tasks repeatedly until they get easier and the pain stops. This approach requires a change of mindset and therefore, like many things DevOps, demands a combination of carrot and stick to ensure adoption.

Build quality in

DevOps borrows ideas both from agile development practices and from Lean process improvement, which focuses on eliminating error and waste and is applied equally to both development and operations. These principles design processes to predict and prevent errors before they happen. Pair Programming and Continuous Integration identify errors before they leave the development team. Continuous monitoring and validation of the process steps helps remove bottlenecks and optimize tasks. People are like electrons in a flow of current – they will take the path of least resistance. Ensuring that your processes support the shortest path yet maintain quality is critical to successful software delivery.

"Done" means released

Dev and QA see their processes and goals as separate from those of other teams. DevOps requires these processes to merge in individuals' minds, so that developers retain a sense of ownership of the code until operations completes the release to end users. Ops needs to be aware of builds as they emerge from Dev and they need to be ready to release immediately. The agile state of "done" from a development perspective – i.e. the story is complete and handed over to the next part of the process – is not suitable in a CD/DevOps model. Done, in this case, means delivered safely to production.

Everyone is responsible for the delivery process

Although the ideal is to flatten hierarchies, break down silos, and integrate teams, flattening the Dev, QA, and Ops silos into a single integrated team is not always feasible, often for skillset reasons. Developer resources who understand operational constraints and architectures are as rare as operational resources who understand application architectures and code. On the other hand, creating a DevOps team from the ground up does not necessarily lead to successful implementation. We have seen dedicated DevOps teams get created and inserted between development and operations, simply introducing another silo into the organization and causing additional problems. Establishing the shared goal of getting the software into production as fast as possible is more likely to lead to success in the short term while resource issues are being addressed.

A simple DevOps mantra of "if I'm awake, you're awake" can easily increase collaboration between teams, although it should not be seen as a way to simply route more work to key individuals. If development is always available, the likelihood of a successful deployment increases. If development is fully integrated into the process all the way through release, the likelihood of success increases even further.

The preceding seven principles provide a summary of the objectives for Continuous Delivery. The next two chapters of this book will focus on the process and tooling to address more basic questions:

- How do you extend the agility realized in development through to the formal QA, information security and operations teams?
- How do you ensure the production readiness of code at all times, allowing feature- or function-based releases?
- How do you gain visibility into the quality of changes in your deployment pipeline?
- How do you feed insight from production back into development?

How long does it take to get a line of code into production?

For most organizations not doing CD with long release cycles, the answer is likely to be "***Months***."

The reader of this guide should consider this question to understand the effort to release code.

Continuous Delivery: Where to Begin

3

All organizations that create business or consumer-facing applications benefit from the adoption of Continuous Delivery practices, but there are a number of prerequisites that need to be implemented in order to be successful:

- **Understanding of Business Objectives** – The digitalization of products and services, and of the channels through which they are marketed and sold, makes it essential that IT understand what really matters to the business. What are the goals and objectives of the business? What is the plan to deliver against those goals? And how do development and IT operations fit within the context of delivery?

  Many IT organizations have skilled individuals whose sole responsibility is to act as a liaison between engineering and the business. These individuals work with the business stakeholders to get their feedback and input. But why simply include a limited subset of resources in this process? The smaller the number of individuals involved, the more likely that a level of bias – intentional or not – will be included in the transformation of the business requirements. When members of the application development and triage teams, operations, production support, and incident and problem management teams are included in the process, a much greater level of understanding and clarity of business requirements can be achieved.

  This does not mean that IT should spend all of its time in concept or planning meetings, but that an increased level of business understanding will lead to a better final deliverable. A single product manager who "knows everything the customer wants" is not going to lead to a successful product. With greater involvement, IT can understand the vision, direction, and objectives of the business and implement a culture that supports rapid and successful delivery of high-quality, objective meeting applications. This integrated approach will often lead to more complete and successful implementations, as the longer-term objectives are included in design, definition, and implementation.

- **Executive Sponsorship** – DevOps, like any other significant initiative, thrives with an executive management champion who has the authority to change business processes. The sponsor does not need to be the CIO, but with the current level of hype surrounding DevOps, the CIO may be more receptive to it. Business pressures alone may mean that the executive sponsor is ready to put an initiative in place to speed up the velocity of releases and enable more frequent application deliveries – and may use the term Continuous Delivery or DevOps to describe it.

- **Continuous Integration** – CI is essentially a development practice, but because it also involves a level of automated software testing (at least unit testing), it begins to stray into the realm of QA. Agile approaches commonly integrate development and testing resources, so teams should not view incorporating unit testing into the development cycle as a new concept. As CI models push to deliver build artifacts further to the right – towards operations teams – the impact upon QA teams becomes more noticeable. Advocates for CD tend to agree that there is little point in attempting to promote DevOps in an organization that has not yet adopted CI and made at least this level of commitment to process automation.

- **Change Management** – Many enterprises are committed to ITSM/ITIL processes for organizing IT delivery and tracking issues. Although adoption of ITIL does mean implementation of formal release management or service transition processes, effective software deployment is reliant upon well-defined operational change management practices, something towards which ITIL provides good guidance. ITIL practices, and DevOps practices in general, often overlook the value of tracking development-level changes against operational change. Having both a well-defined, ITIL-compliant operation process and a well-defined, agile-based development process, with little to no interaction between them, immediately causes a disconnect between Dev and Ops.

  At the enterprise level, there are frequently multiple tools in place; remember that the best-of-breed tools for Ops change and the tool of choice for Dev change may be two completely isolated products with little to no process overlap. A simple way to break down the walls between Dev and Ops is to develop a common change management process for the entire deployment pipeline. It should be noted that change management is a risk-reduction strategy, and many CD advocates view a formal change management process as unnecessary in a successful deployment pipeline. The question must be asked for your organization: Is it enough to get deployments into your environment quickly, or do you also want the ability to validate and explain which specific Dev and Ops changes were delivered as part of the deployment?

- **Automation** – Automating code and configuration deployments with a single set of deployment processes across all environments, using parameterization, will help ensure that environment-specific constraints and requirements are met. Do not deploy code in different ways simply because you are moving from test to production. Repeatability and consistency are the key to successful, accurate, and speedy delivery. Ensure that all artifacts are deployed from the same source location or artifact repository.

  Deploying in the same way across all environments is efficient in both time and cost. Using the same process ensures more consistent testing, and any environment-specific issues are far easier to identify. The more automated this process is, the more repeatable and reliable it will be. This translates into faster deployments, reduced outage time, and more confidence from the business that change is being delivered in a reliable manner.

One way the walls between Dev and Ops can be broken down is by developing a **common change management process** for the entire deployment pipeline. This includes integrated processes for both development and operational changes.

Other tools and processes may create conditions favorable for CD but do not need to be seen as prerequisites. For example, DevOps overlays Application Lifecycle Management (ALM) and Enterprise Service Management (ESM), the chain of disciplines by which organizations manage applications from "cradle to grave": requirements management, software architecture, development, testing, software maintenance, change management, project management, release management, and service management. Some organizations include Product Lifecycle Management (PLM) as well. ALM may have various degrees of process control, and every organization has ALM even if they do not call it that. Continuous Delivery and, in turn, DevOps might be considered an evolution of the notion of ALM (or the ALM 2.0+ definition created by Forrester in 2010[3]). There is no specific implementation of ALM tools or products that must be in place for a DevOps transformation strategy to be successfully adopted.

Continuous Delivery is a compelling first step for virtually any type of enterprise, although proponents of CD tend to advise that it is more suited to ***certain types of applications*** than to others. Indeed, the clamor for DevOps and CD adoption at enterprise levels often fails to take into account that certain applications are architected and implemented in a way that makes CD impractical. Continuous Delivery is highly appropriate for web-based applications or mobile applications, given the volume and velocity of changes common in development of these types of applications. Any enterprise that undertakes Java or .NET application development is very likely to have Continuous Integration or common build processes already in place. Older "legacy" technologies may be less appropriate initial targets for CD; however, many organizations try to prove capability by undertaking the hardest challenge first. This approach often leads to failures or poor rates of adoption and low returns on investment.

Gartner refers to a practice of "Bimodal IT"[4] – agile and continuous for some applications, and stable and sequential for others. In reality, this practice is "multi-modal IT," where applications have various speeds of delivery and change, with the agile and risk averse meeting at various stages, depending upon a number of factors such as organizational maturity and application development team skillset. Applications deemed architecturally unsuitable for CD may be identified as candidates for implementation at a later date.

An enterprise application portfolio can be one of the determining factors for an organization's readiness for DevOps. An important question to ask about a business-critical application is whether it can support this new world of rapid change. If not, that could be a signal that it is time to rework or replace that application with one that is architected for CD, although this can be a costly and time-consuming exercise. Organizations can determine the capability of applications by going through a realistic portfolio analysis, and overlaying that with a detailed business analysis of the value of the application. High business value, low complexity, and well-architected products are a natural starting point for any enterprise CD strategy. Pick the right applications and gain traction first, then spread to the rest of the enterprise at a later time. In this way, success breeds success.

[3] Carey Schwaber et al Forrester 2008-2010 https://www.forrester.com/report/ALM+20_Getting+Closer+But+Not+There+Yet/-/E-RES46166

[4] http://www.gartner.com/it-glossary/bimodal

Continuous Delivery and the deployment process

The key value feature of the CD process is the deployment pipeline. When a developer commit happens, in old parlance when a developer checks in a code change, a CI process will incorporate the change into the rest of the repository-stored code base, and the software gets built and unit tested. A successful build and unit test will instantiate the deployment pipeline – simply a modeled representation of logical environments into which the newly deployed artifacts should be deployed – with this process repeated every time a code change is committed by a developer.

When the new build completes unit testing, the deployment pipeline automatically deploys the code to the next environment, where it will generally undertake some level of integration testing. Once a deployment pipeline has been initiated, automated testing becomes more and more important. If for some reason the build fails, the developers are notified immediately and can start the process of remediation. By identifying issues in the build as soon as a code change is delivered, the amount of time taken to resolve issues with the build is significantly reduced. Developers no longer argue about "who broke the build" and which changes caused the issue, but can identify this immediately (the last commit broke the build). To ensure that quality code is delivered, each build goes through the pipeline and undertakes various levels of automated testing, including performance testing, capacity testing, and security penetration testing. Eventually the build may undergo a manual testing process, usually some form of UAT, that will provide business sign-off. In this case, deployment can be seen as a technical decision to release a feature within the software, whereas a release is associated with a business decision to introduce the feature or features into production or make it available to the end user.

An interesting question, and one that can only be answered by the organization attempting to implement a DevOps strategy, is whether the DevOps initiative should impose an ideal structure and process onto the IT organization, or whether it should align itself with existing processes. The reality is that as long as the prerequisites for adoption are met, it generally does not matter. The process and tool changes required for successful adoption of a DevOps transformation program mean that there is no hard and fast rule – although a reasonable answer might be whichever is most likely to work within your organizational culture.

Once you have implemented Continuous Delivery and have put a deployment pipeline in place, how does your application release process change?

A common misconception implies that CD requires breaking down the siloed organization of IT departments and changing the reporting structure to make it receptive to a single continuous process. In reality, it is unrealistic to expect organizational change to occur immediately, and such a change is not a necessary precondition to undertaking a DevOps transformation or implementing CD. The success of the transformation project will depend on the ability and willingness of the development, test, and deployment teams to collaborate on a single, streamlined process.

Development may have been following agile methodologies for more than a decade and may have mature and well-running processes in place. There are also ways of achieving agility in IT operations, but this is not typical at the enterprise level, especially in U.S. enterprises.

A common approach to agile development, Scrum, allows the developers to participate in the analysis, design, and building of an application, with rapid feedback cycles that often involve Continuous Integration, feedback loops, and automated testing. This allows the development team to produce working or deliverable software at the end of every "sprint" – a time-boxed deliverable period. Unfortunately, in many organizations, test and deploy are done in waterfall fashion, because these phases do not belong to development and the QA and release management teams want to control the ongoing flow of data. Existing silos and barriers are difficult to remove until improvements in quality and velocity can be proven.

In a model based on a Continuous Delivery, the goal is for each sprint to include Analysis › Design › Build › Test › Deploy. It has been proven, first in manufacturing and later in software development, that delivering in smaller batches with continuous flow leads to greater quality and faster delivery. Without the roadblocks or walls in place, the flow of data and applications is much smoother.

The problem is always with the handoffs through the lifecycle once the build is completed. The primary gaps occur between build and deployment. That moment typically is characterized by the handoff to a dedicated test organization that is responsible for managing and provisioning the test environments and then deploying into them. It is during these handoffs that each team assumes that its responsibilities have been completed and loses sight of the overall objectives of the release.

In a traditional model, the handoff from development to operations was handled by project managers or release managers, but the two organizations remained separate. This separation of job function and process will not work in a CD model. Developers must become more closely aligned with operational concerns, and operations teams need to involve themselves in at least downstream aspects of development. That does not mean that operations teams suddenly have to understand complex application architectures, but they should at a minimum understand application dependencies and alignment.

There are challenges to be addressed at all points along the path to production. For example, before the new version of the application can be deployed, the QA team may need to log a change order or ticket with the operations group. This ticket simply states that the application must be deployed into environment x at a certain point in time. That environment could be a single server or multiple servers with many dependencies. In some organizations this process gets repeated for each environment – not using the same ticket but creating a new ticket each time and attaching the correct application versions and deliverables, including the deployment order.

With organizations redeploying code into different environments using a different set of instructions and potentially different binaries, is it any wonder that production deployments are the cause of up to 85%[5] of production issues? By contrast, using a CD model, where the process of deployment to different environments is automated, hopefully to the level of self service, enables the QA team to say, "I want this particular application version deployed to this particular environment at this particular point in time," and that deployment will happen in a fully automated way, removing a resource bottleneck and replacing a potential failure and area of risk with an interaction-free process flow.

[5]Gartner: How IT Operations Can Set Up an Effective, Centralized Release Management Process. Published: 3 June 2013; Analyst(s): George Spafford, Ronni J. Colville.

A true DevOps-centric organization also concerns itself with provisioning the infrastructure for development, the infrastructure for test, and the infrastructure for deployment as well as for integrating these with the application deployment and testing processes. The new process flow and interaction model will in turn drive the adoption of new tooling and process improvement. Ultimately, the deployment pipeline will look more like what is depicted in Figure 1.

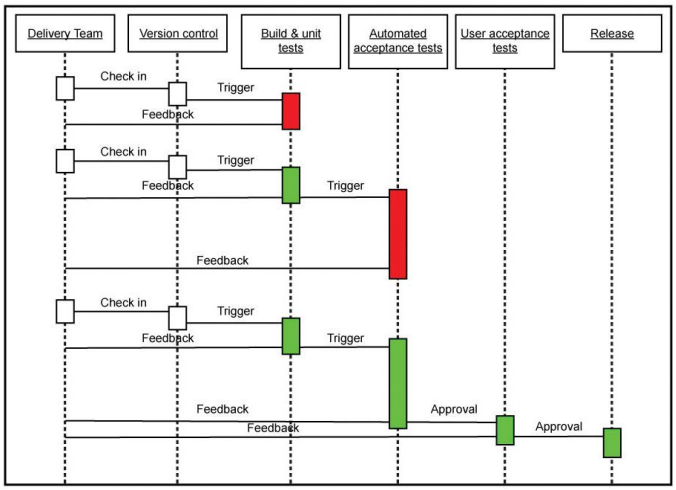


Figure 1: DevOps deployment pipeline

As the graphic shows, there are many opportunities for automation tools to impact the roles of individuals in the deployment process.

The tools that implement Continuous Integration should be the first part of the deployment pipeline. Operations may not have a direct interest in the CI build process, but as this process triggers testing and subsequent deployments, the outcome of the CI build is of direct interest to the eventual deployment process.

IT operations teams are more than familiar with the concept of executing a script against a particular environment. However, the concept of modeling an entire deployment process, featuring automation of the process, could potentially be new to them.

Unit tests are typically automated as part of Continuous Integration, but organizations still struggle to automate integration and acceptance testing. You can't automate manual testing.

Impact on the testing process

The QA function in many organizations is evolving from the traditional waterfall-based, handover-centric approach to a more integrated, fully encompassing process. As development teams have become responsible for increasing the level of unit testing by implementing CI, QA teams have had to adjust to a much faster rate of delivery and availability of newly created or modified features. The unit-tested code produced by development teams will subsequently be deployed – hopefully using automated processes – into an integration test environment, where it will be tested using any number of automated testing tools and then deployed and validated in the UAT environment before final release into production. However, a lack of visibility into this end-to-end process causes process inefficiencies and a lack of real-time traceability and status accounting.

In many cases, the QA organization is responsible for trying to connect the dots between these different environments. Take, for example, the physical unit testing environment, owned and provisioned by development; the integration test and UAT environments owned by the QA team; and the production environment owned by operations. Each team may have its own process for provisioning these environments, which, in an ideal world, would be identical in terms of setup and configuration. To move the code from unit test to integration test, development may assign a change request to the QA team and wait for QA to mark the environment as available or even to provision infrastructure. The subsequent handoff from UAT to production may be similar, but operations' process is its own and may be different from the process followed by QA.

The implementation of DevOps or CD practices will require the QA organization to become much more closely aligned with both the development organization, to ensure a thorough and complete understanding of what is coming down the pipeline, and also the operations team, to ensure that the final handover to production is truly successful. Testing teams need a much greater understanding of many automation technologies, from application release automation tools that deploy the files to the managed environments, to automated testing tools that validate the application, and even to the infrastructure provisioning tools that are used to create new environments and physical infrastructure where needed. The number of environments needed to support feature-based CD also makes the notion of containers and container-based images attractive, increasing the throughput of testing capacity in line with increases in development.

4

Automation of the Deployment Pipeline

As we saw in Chapter 1, Continuous Delivery advocates argue for the automation of as many processes as possible, from one end of the release process to the other. Automation that supports effective development processes (e.g. peer review, static analysis, and unit testing) ensures higher quality of development at the beginning of the deployment pipeline.

There is a simple rationale for this: people should not move or deploy the "bits." Systems are better and more consistent at deploying applications than humans. Automating manual tasks and handoffs is one of the first things you should look at on the road to Continuous Delivery. You can achieve quick wins with automation, and this approach can be undertaken in a way that targets the easiest or most valuable applications first, and is then repeated rapidly and without major organizational changes.

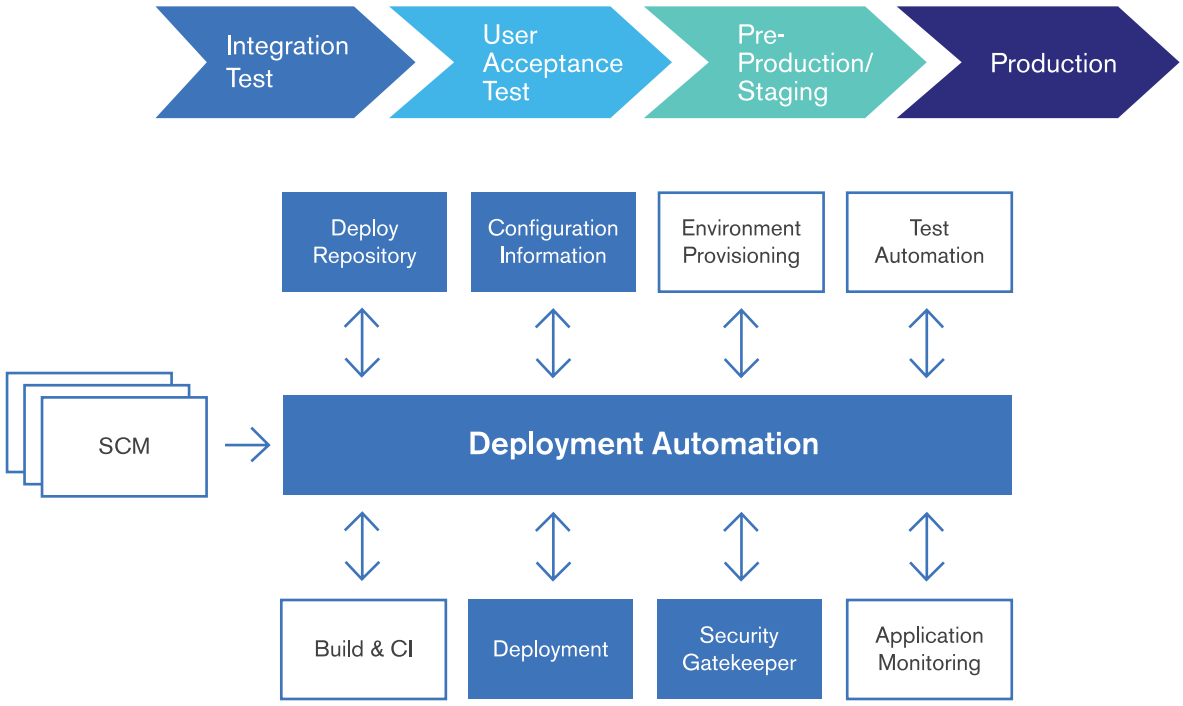


Figure 2: Toolchain integration using Micro Focus' deployment automation topology

Through shared technology solutions, development and operations team members can work together to increase the velocity of change through the system, enforce best practices, ensure compliance, and ensure that releases are repeatable, reliable and predictable. As discussed earlier, the earlier development failures are caught or identified, the more likely the team is to eliminate rework, improve quality and ensure release readiness of the code.

Complementary task-specific tools are used in conjunction with capable source, build and deployment tools to automate the deployment pipeline. This so-called DevOps toolchain may incorporate many tools from different vendors and use different integrations within the release and deployment process. Some of the principal tools in the DevOps toolchain include:

- **Source Code Management Tools** – Ideally with integrated change management, work item management, bug tracking, or request-handling capabilities, source code management tools allow multiple feature-based deliveries to be tracked and approved.
  These tools will integrate fully into the appropriate build/CI tools.

- **Continuous Integration (CI) Tools** – Intended to monitor for changes to the code base and manage its automatic rebuilding and unit testing, CI tools automatically notify the developer should the build or unit test fail. CI tools may include artifact management integration capabilities.

- **Continuous Delivery (CD) Tools** – Providing an automated deployment mechanism usually triggered without human intervention to deploy code to the next area in the deployment pipeline, CD tools give feedback to stakeholders at all touch points in the release process.

- **Deployment Pipeline Automation Tools** – Engineered to move digital artifacts through development, test, and production in a predefined or dynamic order, pipeline automation tools should allow users to track the contents of environments at any given time.

- **Release Management Tools** – Designed to define, manage, and track a release through its end-to-end process, this includes the management of physical artifacts of the release, relationships between applications, services, platforms, or even components, and the changes included in a release, both from a business and a development perspective. Other important features include links to test cases, activity sequencing, approvals, and milestone management.

- **Monitoring Tools** – The deployment toolchain is typically integrated with Application Performance Monitoring applications that track the impact of any newly deployed code and allow operations teams to identify any anomalies as soon as they occur. These tools frequently include some extended analytics functionality and performance dashboards.

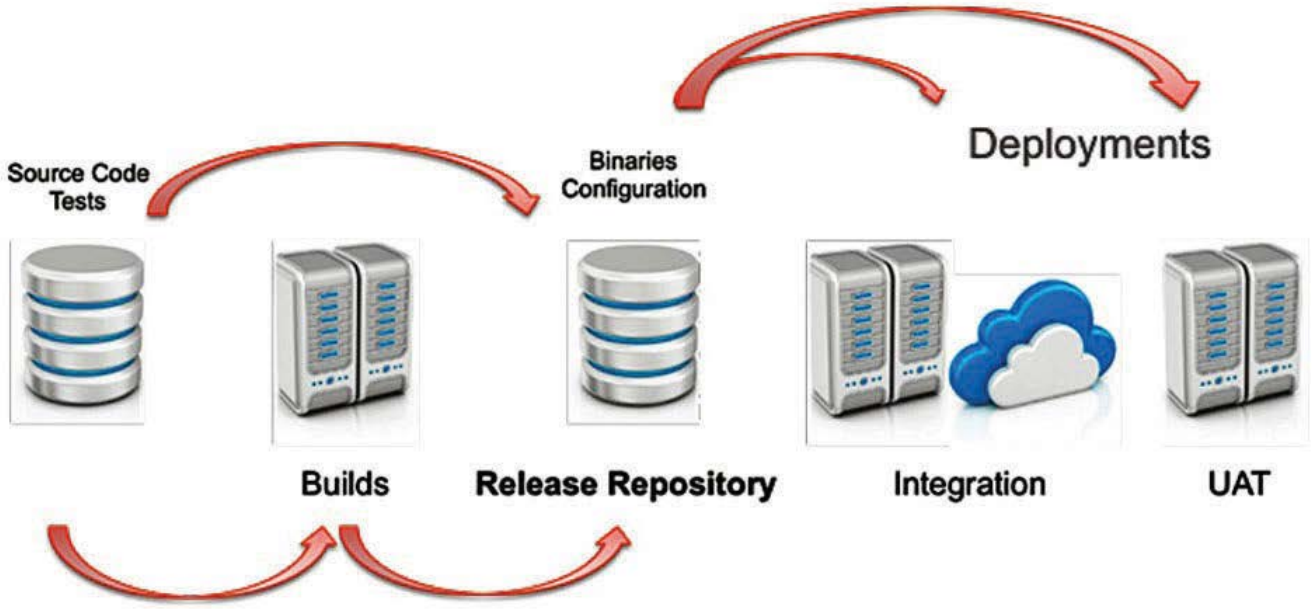


Figure 3: The release repository ensure that deployed componenets are identitcal to those testing in preproduction.

The core process: Continuous Integration and Continuous Deployment

The CI process is typically owned by the development organization. Every change to a software code base is essentially tested in real time. When an individual commits a code change, this triggers an automated process to test that software so that, at the end of this process, the enterprise has something that is deployable. The intended result is fewer defects in the code released for deployment.

The build process is triggered based on the developer commit, with any developers who cause a build failure getting immediately notified (by the build tool) that they broke the build. In this scenario, it is the responsibility of the developer who breaks the build to fix any errors and get the build working again. If more than a single build failure occurs, escalation and additional notifications may be generated to notify the rest of the team that the build is unstable.

As part of the CI process, some form of automated testing is set up to run in the background and is triggered as developers commit and build their code. This automated test suite will continuously validate the compiled code and report any test failures to the developer. The actual level of validation can vary from organization to organization, but it generally includes some form of API or functional interface testing (these are often the easiest test suites to automate). The extended testing process may include developers initiating the automated test suite to run after a successful build or on a schedule at the end of the day as part of a nightly build process. This automated testing suite can help determine whether all the components that multiple developers have been working on can be integrated together as an entire application. Simply speaking, this is just an automated form of integration testing.

The automated test suite will consist of many thousands of test cases launched against the code base. In an ideal world, the code base would achieve 100% unit test coverage.

The basic building blocks of the code base of your application (like the RESTful services and API calls) should be prioritized for automation, with extended logic and business functions following afterwards. If any test fails, the developer is automatically notified and is responsible for rectifying or backing out the problematic change.

Once automated unit testing is successfully completed, the code should be automatically deployed to the next test environment, where additional testing types may occur. A percentage acceptable failure rate may be deemed appropriate; this is the decision of the development lead, as that individual has ultimate responsibility for the code base and the compiled code that is being delivered to the next environment. Once automated testing completes successfully in the next environment, automated promotion to additional test environments may occur. The process simply repeats with automated application deployments and varying types of testing being performed against each environment. ***The goal is to have a repeatable, automated process to reduce human error and take away the risk of manual interaction.***

Continuous Deployment can be used to deploy applications all the way to production, but operations teams are generally uncomfortable automating production deployments unless they have been involved in the creation and prior running of such deployments. Much of the time this means that Continuous Deployment ends once the code reaches a preproduction staging area, and then traditional manual deployments or existing scripts are used for the last step to production. While this approach is understandable and cautiousness about production deployments is laudable, it perpetuates the basic problem: different tools and scripts applying changes to your key production environments that were not used in testing, and were not themselves tested, causes significant outages and subsequent organizational embarrassment.

For many IT leaders, it is difficult to distinguish between Continuous Deployment and Continuous Delivery. To understand the difference, we should think about the conventional delivery pipeline. In traditional development and delivery approaches, there are controls or gates at each point in the path to production that demand an affirmative action in order to continue. Think of the handover and validation steps of a traditional software development lifecycle, which may include approvals to enter and exit every environment.

If a Continuous Delivery approach is implemented, the application should always be "deployment ready" and could in theory be delivered into production at any time. Often organizational rules or practices are in place that prevent this from happening, such as concerns about segregation of duties or unauthorized production changes. With Continuous Delivery, automated deployment and testing processes are infrequently put in place for lower-level pipeline environments such as development and integration test, but are not allowed to run in the higher testing, staging, and production environments. The end goal is to always deliver code that could go to production, even if it isn't going to.

If a Continuous Deployment approach is implemented, code is always production ready, and, as long as the deployment and test automation processes are successful, the code deployment automatically continues through all environments and eventually into production, with zero human intervention or oversight along the way. This potentially allows zero-touch deployment from developer commit through build, deploy, and test and into production.

Using feature-based development (feature flags), CI can ensure that the code base is releasable at any point in time and that the delivery of code can be based on a business decision, as opposed to an operational constraint.

Let's look at the various classes of tooling for the deployment pipeline in more detail.

Deployment pipeline automation tools

These tools automate the delivery and validation of code through its lifecycle. They automate the delivery of change, potentially enabling a greater volume of releases to be deployed into production. The rapid delivery and validation of deployed code through the use of automation can help reduce error rates and facilitate predictable and reliable releases.

Deployment pipeline automation tools impact the delivery of applications in multiple ways. For the application being deployed, these tools provide:

- Access to the source and executable code repositories, the configuration files, and data elements to be deployed
- A defined process for deploying an application that can be modified easily and that makes use of common elements for standard tasks (such as starting and stopping a web server or backing up a database)
- Ability to define a controlled sequence of target deployment environments
- Support for "compensating transactions" for fully automated rollback
- The ability to invoke a chain of events that can potentially provision the entire application "stack," including infrastructure, application, and configuration

Lack of suitable test environments, aligned with contention in the use of environments, can delay deployments and increase the cost of releases. True Continuous Delivery requires the availability of test and preproduction environments – there is no point in having an application compiled and ready to deploy if the environment into which it will be deployed is still being used by the previous delivery. The use of infrastructure provisioning tools, allowing new environments to be created on the fly, is a great way to work around such constraints. By creating new physical, virtual, cloud, or container environments as part of the deployment and pipeline process, organizations are able to reduce deployment cycle times significantly.

Additional visibility into both the current and proposed use of environments can be exposed by a calendar-type view, which can also provide a single location for a unified test management schedule. Getting visibility into who and what is using, or is scheduled to use, a set of environments provides a level of insight that many enterprise organizations struggle to attain. The schedule view will also show any scheduled deployments, any scheduled maintenance periods, and the available release windows for each environment.

For target environments, deployment automation tools offer:

- Access and visibility into test and production deployment environments
- The ability to create new physical, virtual, cloud, or container-based environments as needed
- A method for physically distributing the application to the target environment and to invoke associated test suites

For the deployment engineer, automation tools provide:

- A simple CD service driven from the CI tool
- The ability to perform one-click deployments, either on demand or on a scheduled basis
- Real-time monitoring of deployment and automated test progress, with the ability to respond to intervention requests
- A dashboard with all historical data, previous deployment times, trends, patterns, and currently deployed application and component version information

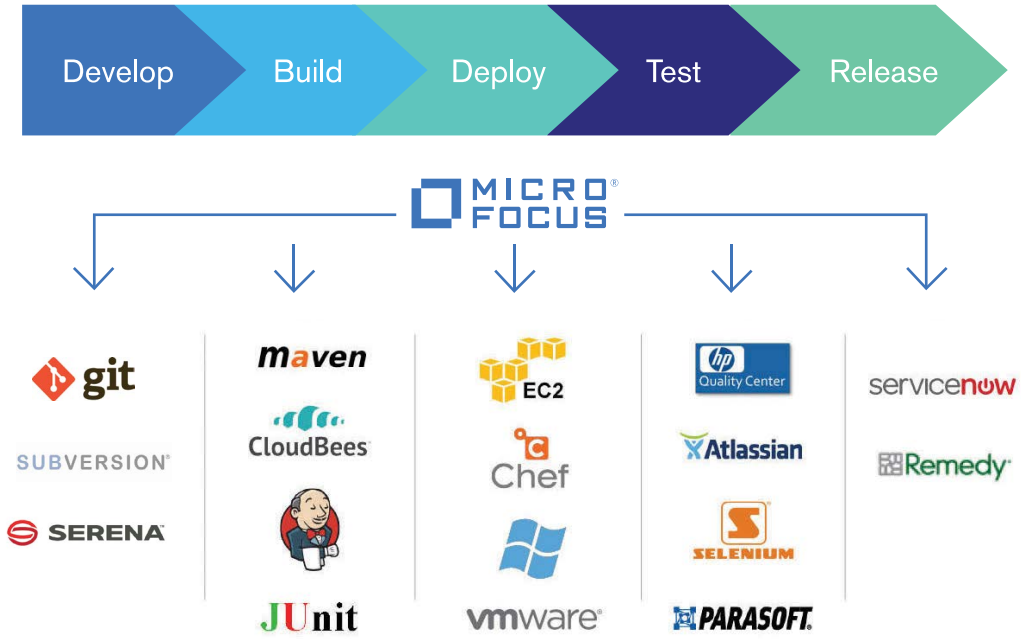


Figure 4: Micro Focus DevOps toolchain integration

Deployment automation has been proven to reduce deployment execution times by over 90% and error rates, rework, and failed deployments by as much as 90%. As release engineers gain expertise in the use of these products, they will be able to automate additional application deployments (through the reuse of existing processes) in much less time than it takes to create application-specific deployment scripts.

An effective deployment automation tool will integrate with CI, provisioning, deployment, and automated testing, giving you the ability to schedule deployments and automate ad hoc deployments to any environment.

While a common set of tools across organizations may be desirable, many times the development, QA, and operations teams are already bound to existing tools that cannot be replaced easily or quickly. In this case, it is important to have the flexibility to knit these tools together into a coherent toolchain. Again, the use of public APIs or, at worst, of command-line interfaces will allow the products to communicate. In this case, a single, central point of visibility or insight into the overall pipeline process will be needed.

As part of the release process, one of the most important areas of integration is ensuring that the release management tool can integrate with the change management system. This allows different members of the organization to gain insight and visibility into both the change and release process. In many organizations, specific enterprise change management tools are in place – for example ServiceNow or BMC Remedy. While these tools provide great high-level process and approval management, their lack of integration to artifacts or development- or test-centric tooling (agile planning tools, artifact repositories, or build tools) means that a large disconnect exists between formal change and release processes. Naturally, the ideal scenario is to be able to provide complete traceability from the change request to the deployable artifact. Updates that occur in either or both systems need to be reflected in the other, and the solution needs to drive state transitions between the change and release processes (or vice versa). To support this true change management and release management integration, a tool that can orchestrate either itself or other tools and processes based on the state of the change or release process provides a significant advantage.

An integrated platform for Continuous Delivery

Deployment Automation[6] (DA) is a single platform supporting the entire deployment pipeline. The deployment process spans multiple environments and integrates seamlessly with other products as part of a true DevOps toolchain. Critical tool integrations such as those between development tools (software configuration management tools, build server/CI server), provisioning tools, deployment automation, and manual or automated testing tools are provided out of the box. The extensible plug-in architecture allows enterprises to orchestrate these processes and integrate with the various tools.

[6] https://www.microfocus.com/products/deployment-automation/

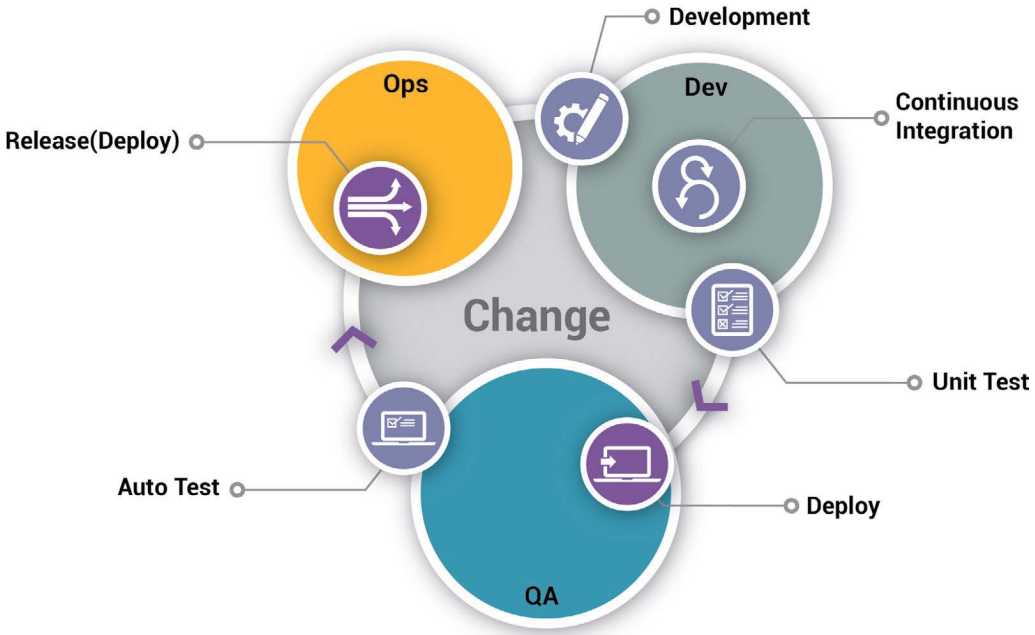


Figure 5: Integrated platform for CI

In a sample use case, a developer commits code to the source code repository in response to a valid change request. The CI tool is invoked automatically and the system produces a new build. Automation in Micro Focus' market-leading Application Lifecycle Management (ALM) product, Dimensions CM, will ensure that any changes included in the build have passed both peer review and static analysis and have been successfully unit tested. If the build process is successful, the artifact is uploaded into MicroFocus' secured artifact repository, and SDA will automatically deploy the new version of the component or application to the first environment defined as part of the deployment pipeline.

The deployment automation technology will create and configure any required infrastructure prior to the deployment and also configure the application as needed. SDA will invoke the test automation tool, which will run automated tests and will log whether the testing suite has passed or failed. If the test suite fails, the system will automatically roll back the application to the last version that was deployed successfully, and will report to the developer and test lead that the application did not pass automated testing. If the application test suite runs successfully, SDA will automatically deploy the application to the next environment in the pipeline.

As part of the deployment process, the application may need to be configured and deployed to an application server space (e.g. WebSphere). A simple process to perform this would be to stop the application server, deploy the updated application version, and restart the application server, backing up existing files, renaming .war files, etc. In certain circumstances, you may need to provision the environment first and then deploy. SRA can perform this entire sequence automatically.

Outside of the application pipeline and delivery process, SDA integrates seamlessly with Release Control[7]. RC provides a fully featured integration framework that is integrated with change management ticketing systems (for example, ServiceNow or Remedy), and the successful deployment can trigger an update to the change management ticket, allowing those not directly involved in the release process but tightly aligned with the change process (business users, for example) to have real-time insight into the progress of the change and its associated release.

[7] https://www.microfocus.com/products/release-control

As part of the DevOps transformation initiative, many organizations choose to limit “repository sprawl” and create a centralized “hardened source code management system.” As open source software has become increasingly tolerated and even encouraged, the number of source code repositories has grown exponentially. Technology that was once cool (like the Subversion version control system) is no longer seen as the version control software of choice. Developers are now moving to Git-based repositories, but they rarely migrate their histories and versions from SVN. This repository sprawl can have a serious impact on the safety and security of an organization’s intellectual property. Having the code base in multiple systems is not suitable, especially when audit and compliance are involved.

A software change and configuration management (SCCM) system that provides the best of both worlds – offering a user experience similar or identical to Subversion or Git, while also providing a centralized and hardened environment (configured securely, administered correctly and installed on a secure server) – is the ideal solution to reducing enterprise repository sprawl. Not all SCCM products provide the levels of security and control that modern enterprises demand. Many suffer from a lack of fine-grained and integrated access controls, detailed auditing and logging (to identify anomalous behavior, provide auditability, and underpin forensics post incident), integrated peer review, and code baseline management.

In summary, deployment pipeline automation to support DevOps and Continuous Delivery can be piecemeal, or it can be through an integrated platform. The decision to include a single platform or multiple disconnected toolsets can be driven by budget, process maturity, or organizational culture, but the end result – to support a DevOps or CD transformation program – should always be taken into full consideration.

5 Conclusion

DevOps transformation programs and the implementation of Continuous Delivery practices can significantly reduce your time to market. They can also increase the quality of the applications you build within your enterprise – and, through these improvements, increase the value of the services you deliver to your customer. They can dramatically reduce your development costs. However, DevOps or Continuous Delivery practices can be hugely challenging to adopt at enterprise scale. These practices may require process changes that are unsettling to your developers, testers, and operations team. Buy-in may not be unanimous; in transformation initiatives, it rarely is.

Many organizations have begun to implement Continuous Integration, which is a step towards Continuous Delivery. However, a large number of them have failed to transition from one to the other and are unable to automate their deployment and release processes. This may be attributable to differing objectives and performance metrics at the opposite ends of the application lifecycle. Overcoming such differences generally requires strong top-down leadership, but thankfully it rarely requires an overhaul of the management hierarchy across the IT organization.

True adoption of CD will be easiest in enterprises that have already implemented agile development processes. Ideally, CI process automation will play a highly significant role in any DevOps initiative, but it will not be sufficient to bring about the transformation in and of itself. Tool and product implementation alone will not drive organizational change and ensure the successful adoption of a new way of working across multiple organizational units.

CD can have a significant influence on the agility, and thus the competitiveness, of an enterprise – not just the speed of application deployment. In the modern business climate where products and services are increasingly digitalized and where applications are no longer extensions to the business, but are frequently core to the business itself, it is essential that the stakeholders in any DevOps initiative view it as a transformation of the entire enterprise and an opportunity to enable the business to drive significant competitive advantage as well as market and thought leadership.

However, even in enterprises where agile processes have given developers more direct exposure to business requirements, Dev and Ops are frequently just beginning to understand what it is that the business actually does. Conversely, the business is just beginning to see the value in bringing corporate IT up to speed on enterprise strategy and to understand the value technology really contributes. IT is no longer merely providing the infrastructure to manage batch processing transactions (although this critical role is still undertaken and needs to be managed as a critical business function) but is now actually "the business." Think of this from a consumer perspective; you are much more likely to visit a financial institution from a smartphone than to visit a branch in person. The "face" of the company and its brand is now the 5" x 3" screen experienced on a train, on a bus, or in an office, as opposed to the well-trained and experienced branch staff member. IT has become the portal of B2C interaction and needs to be thought of as such.

Thus, while software development teams have widely adopted agile methodologies, their enterprises are just starting to appreciate the value this creates and how this can lead to better interaction with the customer base. A satisfied customer is much less likely to look elsewhere for similar services. Naturally, when an organization interacts with high-net-worth customers, the experience has to be as smooth as possible. To misquote a famous novel, "All customers are equal, but some are more equal than others."

Enterprise executives realize that agile development is insufficient to drive improvement in application time to market, as development is only the front end of the application delivery pipeline. In many organizations, counterproductive walls between development, QA, and operations still exist, and while development practices have become more agile, testing and deployment are still stuck in a more of a batch-and-queue mode, clinging to their traditional waterfall approaches.

Continuous Delivery and DevOps transformations typically begin with a process review and the automation of the deployment pipeline. While this is a gallant approach, a more fundamental problem must be addressed: development and IT operations people have different approaches, concerns, and performance metrics, and they frequently do not understand each other's processes. Dev is from Mars and Ops are from Venus (which may explain some of the strange sights you have seen in IT operations teams). Success in accelerating application delivery will depend not only on bridging these differences in approach, but on eliminating the issues that occur in the handoffs from one silo to another when, traditionally, code is "thrown over the wall" from development to test and from test to deployment.

In many instances, the issues start even earlier in the process, with business throwing vague requirements over the fence to development, which implements them to the best of its understanding and then throws them over the wall to testing, which struggles to validate what has been delivered. In the seams between these process phases, each team is working to meet its own objectives and to score well against its own Key Performance Indicators (KPIs), and it may actually be incentivized not to take ownership of or help accomplish the objectives of the other teams in any way. Competition between silos is all too familiar in large enterprises, where political wrangling and power positioning are more common than you would expect.

Development and IT operations are often just beginning to understand what the business actually does.

The world of Continuous Delivery promotes a different ideal: unifying everyone as part of a deployment pipeline with end-to-end responsibility for the entire process. Individuals retain their technical roles, but developers are accountable for the impact of code changes on the testing and deployment process, and operations people track and anticipate the deployment schedule of emerging code from development. Many years ago, a similar model was put into effect at a large financial institution. Developers were paid bonuses based on the number of changes delivered over the space of a week. Weighting was given to the complexity of change as well as rate of delivery. Naturally, many developers started to increase their rate of delivery, and QA was swamped with poorly written, buggy code.

Seeing the problem with this approach, the development manager introduced a "tax" on any defects delivered by developers and found by QA. The rate of change stayed relatively constant, but the level of code quality was significantly increased: process ownership and interaction had moved from simply development to development and testing, allowing the organization in question to make significant strides into an emerging market. The logical extension of this process is to extend tracking and delivery into the Ops space – and to become a DevOps-centric organization.


MICRO
FOCUS