The Fastest Way to Continuous Delivery

Instantly connect CloudBees® DEV@cloud™ (Jenkins in the cloud) or Jenkins Enterprise by CloudBees (on premise) to XebiaLabs Deployit to immediately reap the benefits of delivering continuously. Set up in minutes—and get faster application delivery plus complete reliability and insight.

Automated Continuous Integration with Jenkins and CloudBees

CloudBees JENKINS

✓ Jenkins on-prem support
✓ Free & enterprise plugins
✓ Jenkins in the cloud with DEV@cloud

COMMIT → **BUILD** → DEPLOY → **TEST** → DEPLOY → **STAGE** → DEPLOY → **PRODUCTION**

XebiaLabs
XDeployit

Automated Deployment with XebiaLabs Deployit

✓ Agentless, fully automated deployment
✓ Auto-rollback & configuration analysis
✓ End-to-end integration & audit trail

CONTINUOUS DELIVERY IS EASY – TRY IT FREE AND SEE!

- Jenkins in the cloud or Jenkins Enterprise by CloudBees (on-prem): www.cloudbees.com/jenkins/CCD-ref.cb
- XebiaLabs Deployit for instant, automated deployments: www.xebialabs.com/trial

DZone Refcardz

CONTENTS INCLUDE:

Preparing for Continuous Delivery

By: Benjamin Wootton

ABOUT CONTINUOUS DELIVERY

Continuous delivery is a set of patterns and best practices that can help software teams dramatically improve the pace and quality of their software delivery.

Instead of infrequently carrying out relatively big releases, teams practicing continuous delivery aspire to deliver smaller batches of change into production, but much more frequently than typical - weekly, daily, or potentially multiple releases per day.

This Refcard explains this in more detail, giving guidance, advice, and best practices to development and operations teams looking to move from traditional release cycles towards continuous delivery.

OBJECTIVES

Continuous delivery should help you to:

- Deliver software faster and more frequently, getting valuable new features into production as early as possible;
- Increase software quality, system uptime and stability;
- Reduce release risk and avoid failed deployments into both test and production environments;
- Reduce waste and increase efficiency in the development and delivery process;
- Keep your software in an almost constant production-ready state such that you can deploy whenever you need.

PREREQUISITES

To get there, however, you may need to put the following into place:

- Development practices such as automated testing;
- Software architectures and component designs that facilitate more frequent releases without impact to users;
- Tooling such as source code management, continuous integration, configuration management and application release automation software;
- Automation and scripting to enable you to repeatedly build, package, test, and deploy your software with limited human intervention;
- Organizational, cultural, and business process changes to support continuous delivery.

THE KEY BUILDING BLOCK OF CONTINUOUS DELIVERY: RELEASE AUTOMATION!

Though it is quite valid and realistic to have manual steps in your continuous delivery pipeline, automation is central in speeding up the pace of delivery and reducing the cycle time.

After all, even with a well resourced team, it is not viable to build, package, compile, test, and deploy software many times per day by hand, especially if the software is in any way large and complex.

Therefore, the overriding aim should be to increasingly automate away more of the pathway between the developer and the live production environment.

Here are some of the major areas you should focus your automation efforts on.

Automated Compilation & Packaging

The first thing you will need to automate is the process of compiling and turning developers' source code into deployment-ready artifacts.

Though most software developers make use of tools such as Make, Ant, Maven and NuGet to manage their builds and packaging, many teams still have manual steps that they need to carry out before they have artifacts that are ready for release.

These steps can represent a significant barrier to achieving continuous delivery. For instance, if you release every three months, manually building an installer is not too onerous. If you wish to release multiple times per day or week however, it would be better if this task were fully and reliably automated away.

> **Best Practices:**
>
> Implement a single script or command that enables you to go from version controlled source code to a single deployment ready artifact.

Automated Builds & Continuous Integration

Continuous integration is a fundamental building block of continuous delivery.

This involves combining the work of multiple developers and continually compiling and testing the integrated code base such that integration errors are identified as early as possible.

Ideally, this process will build on the previous step so that your continuous integration server is continually emitting a deployment artifact containing the integrated work of the development team, with each outputted build being a viable release candidate.

Typically, you will set up a continuous integration server such as Jenkins, TeamCity, or Team Foundation Server, carrying out the integration many times per day.

Third party continuous integration services such as CloudBees DEV@ cloud, which provides Jenkins as a service, can help to expedite your continuous delivery efforts.
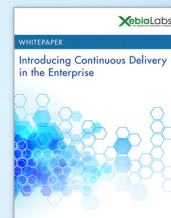
By outsourcing your continuous integration platform, you are free to focus on your continuous delivery goals, rather than on administration and management of tools and infrastructure.

Best Practices:
Implement a continuous integration process that continually outputs a set of deployment-ready artifacts.
Evaluate cloud-based continuous integration offerings to expedite your continuous delivery efforts.
Integrate a thorough audit trail of what has changed with each build through integration with issue tracking software such as Jira.

Your continuous integration process will likely be central for your continuous delivery efforts. For instance, it can go beyond builds and into testing, deployment, and release management. For this reason, continuous integration is a key element of your continuous delivery strategy.

Automated Testing
Though continuous delivery can (and frequently does) include manual exploratory testing stages performed by a QA testing team or end-user user acceptance testing, automated testing will almost certainly be a key feature in allowing you to speed up your delivery cycles and enhance quality.

Usually, your continuous integration server will be responsible for executing the majority of your automated tests in order to validate every developer check-in.

However, other automated testing will likely subsequently take place when the system is deployed into test environments, and you should also aim to automate as much of that as possible.

Your automated testing should be detailed , testing multiple facets of your application:

Test Type	To Confirm That:
Unit Tests	Low level functions and classes work as expected under a variety of inputs.
Integration Tests	Integrated modules work together and in conjunction with infrastructure such as message queues and databases.
Acceptance Tests	Your application and key user flows works as a complete black box when driven via the user interface.
Load Tests	Your application performs well under simulated real world user load.
Performance Tests	The application meets performance requirements and response times under real world load scenarios.
Simulation Tests	Your application works in device simulation environments. (This is growing in importance in the mobile world where you need to test software on diverse emulated mobile devices.)
Smoke Tests	Tests to validate the state and integrity of a freshly deployed environment.
Quality Tests	Application code is high quality –identified through techniques such as static analysis, conformance to style guides, code coverage etc.

Ideally, these tests can be spread across the deployment pipeline, with the slower and more expensive tests occurring further down the pipeline in environments that are increasingly production-like as the release candidate looks increasingly viable. The aim should be to identify problematic builds as early as possible in order to avoid re-work , identifying problems early and keeping the cycle time fast and feedback as early as possible:

Best Practices:
Automate as much of your testing as possible.
Provide good test coverage at multiple levels of abstraction against both code artifacts and the deployed system.

Best Practices:
Distribute different classes of tests along your deployment pipeline, with more detailed tests occurring in increasingly production-like environments later on in the process, all whilst still avoiding human re-work.

Automated tests are your primary line of defense in your aim to release high quality software more frequently. Investing in these tests can be expensive up front, but this battery of automated tests will continue to pay dividends across the lifetime of the application.

Automated Deployments
Software teams typically need to push release candidates into different environments for the different classes of testing discussed above.

For instance, a common scenario is to deploy the software to a test environment for human QA testing, and then into some performance test environment where automated load testing will take place. If the build makes it through that stage of the testing, it might later be deployed to a separate environment for UAT or beta testing.

Ideally, the process of reliably deploying an arbitrary release candidate into an arbitrary environment should be as automated as much as possible.

If you want to operate at the pace that continuous delivery implies, you are likely to need to do this many times per day or week, and it's essential that it works quickly and reliably.

Application release automation tools such as Xebia Labs' Deployit can facilitate the process of pushing out code into environments., . DeployIt can also provide self-service capabilities that allow teams to pull release candidates into their environments without requiring development input or having to create change tickets or wait on middleware administrators.

This agility in moving software between environments in an automated fashion is one of the main areas where teams new to continuous delivery are lacking, so this should also be a key focus in your own preparations for continuous delivery.

Best Practices:
Be able to completely roll out an arbitrary version of your software to an arbitrary environment with a single command.
Incorporate smoke test checks to ensure that your deployed environment is then valid for use.
Harden the deploy process so that it can never leave environments in a broken or partially deployed state.
Incorporate self-service capabilities into this process, so QA staff or business users can select a version of the software and have that deployed at their convenience. In larger organizations, this process should incorporate business rules such that specific users have deployment permissions for specific environments.
Evaluate application release automation tools in order to accelerate your continuous integration efforts.

Managed Infrastructure & Cloud
In a continuous delivery environment, you are likely to want to create and tear down environments with much more flexibility and agility in response to the changing needs of the project.

If you want to start up a new environment to add into your deployment pipeline, and that process takes months to requisition hardware, configure the operating system, configure middleware, and set it up to accept a deployment of the software, your agility is severely limited and your ability to deliver is impacted.

Taking advantage of virtualization and cloud-based offerings can help here. Consider cloud hosts such as Amazon EC2, Rackspace and platform-as-a-service providers such as CloudBees to give you flexibility in bringing up new environments and new infrastructure as the project dictates.

Cloud can also obviously make an excellent choice for production applications, giving you more consistency across your development and production environments than previously achieved.

Best Practices:

Cater for flexibility to your Continuous Delivery processes so you can alter pipelines and scale up or down as necessary.
Implement continuous delivery infrastructure in the cloud, giving you agility in quickly rolling out new environments, and elasticity to pause those environments when there is less demand for them.

Infrastructure As Code

A very common class of production incidents, errors, and rework happens when environments drift out of line in terms of their configuration, for instance when development environments start to differ from test, or when test environments drift out of line with production.

Configuration management tools such as Puppet and Chef can help you to avoid this by defining infrastructure and platforms as version controlled code, and then having the environments built automatically in a very consistent and repeatable way.

Combined with the cloud and outsourced infrastructure, this cocktail allows you to deploy accurately configured environments with ease, giving your pace of delivery a real boost.

Vagrant is a tool which also helps here, giving developers very consistent and repeatable development environments which can be virtualized and run on their own machines.

These tools are all important because consistency of environments is a huge enabler in allowing software to flow through the pipeline in a consistent and reliable way.

Best Practices:

Implement configuration management tools, giving us much more control in building environments consistently, especially in conjunction with the cloud.
Investigate Vagrant as a means of giving developers very consistent local development environments.

Automated Production Deployments

Though most software teams have a degree of automation in their builds and testing, the actual act of deployment onto physical production servers is often still one of the most manual processes for the typical software team.

For instance, teams might have multiple binaries that are pushed onto multiple servers, some database upgrade scripts that are manually executed, and then some manual installation steps to connect this together. Often they will also carry out manual steps for the startup of the system and smoke tests.

Because of this complexity, releases often happen outside of business hours. Indeed, some unfortunate software teams have to perform their upgrades and scheduled maintenance at 3am on a Sunday morning in order to cause the least disruption to the customer base!

To move towards continuous delivery, you'll need to tackle this pain and slowly script and automate away the manual steps from your production release process such that it can be run repeatedly and consistently. Ideally, you will need to get to the stage where you can do this during business hours while the system is in use. This may have significant consequences for the architecture of your system.

To make production deploys multiple times per day whilst the system is in use, it's important to ensure that the process is also tested and hardened so you never leave the production application in a broken state due to a failed deploy.

Best Practices:

Completely automate the production deploy process such that it can be executed from a single command or script.
Be able to deploy the next version of the software while the production system is live, and switch over to the new version with no degradation of services.
Be able to deploy to production using exactly the same process by which you deploy to other environments.

Best Practices:

Implement the best practices described below, such as canary releasing, rollback, and monitoring in order to enhance stability of the production system.

IMPLEMENTING A DEPLOYMENT PIPELINE

Deployment pipeline is a simple but key pattern that gives you a framework towards implementing continuous delivery.

The pipeline describes:

- The explicit stages that software moves between on its path from source control to production;
- Which stages are automated and which have manual steps;
- What the criteria are for moving between pipeline stages, capturing which gateways are automated and which are manual;
- Where parallel flows are allowed.

Importantly, the deployment pipeline concept gives us visibility into the production readiness of our release candidates.

For instance, if you know that you have a release candidate in UAT and a release candidate just about to pass performance testing with a certain set of additional features, you can use this to make decisions about how, when, and what to release to production.

Step 1 - Model Your Pipeline

The first step in putting together a deployment pipeline is to identify the stages that you would like your software to go through to get from the source control repository into production.

The typical software development team will have a number to choose from, some of which are automated and some of which are manual:
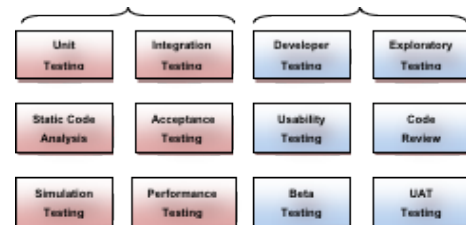
While implementing continuous delivery, you may wish to take the opportunity to add in stages above and beyond those that you do today. For instance, perhaps adding automated acceptance testing would reduce the scope of manual testing required, speeding up development cycles and increasing your potential for continuous delivery. Perhaps adding automated performance testing or manual user beta testing will allow you to shorten your cycle time still further and release more frequently.

Having identified which stages are important to you, you should then think about how to arrange the stages into an ordered pipeline, noting the inputs and outputs of each phase. A very simple example of a pipeline might look like this:
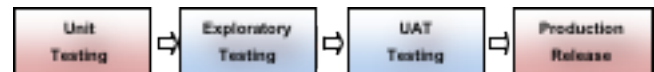

Every software team does, however, do things in a subtly different way. For instance, depending on your comfort with and levels of automated testing, you may decide to skip any form of exploratory testing and rely completely on automated testing processes, reducing the length of the pipeline to a very short fully automated process:

Other teams might choose to parallelize flows and testing stages. This is especially useful where testing is manual and stages are time consuming a fairly likely prospect at the very outset of your continuous integration journey.

Running performance tests at the same time as UAT might be one example of this parallelization. This can obviously speed up the end to end delivery process when things go well, but could lead to rework if one half of the pipeline fails and the release candidate is rejected:
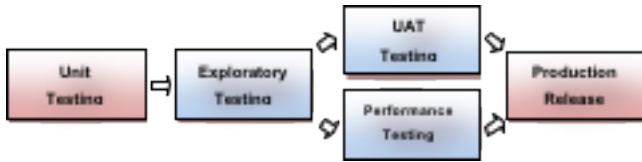
The pipelines above are extremely simple, but illustrate the kind of decisions you will need to make in modeling the flow and implementing your pipeline. The best pipeline isn't always obvious and requires tradeoffs:

| Ideal Situation	Tradeoff
All stages and gateways would be automated.	Requires substantial investment in automated tests and release automation.
Always avoid expensive human re-work.	You need to parallelise test phases if they are slow and manual, giving the risk of failed release candidates at another stage in the pipeline.
Always perform automated testing up-front to reject unviable release candidates	Detailed automated testing is also expensive in production like environments.
Implement lots of environments to support testing phases.	Maintaining environments has a management and financial cost associated.

Whatever position you take on the various tradeoffs, the output of your deployment pipeline modeling should be a basic flow chart that documents the path that your software takes on its journey from source code to production.

Step 2 - Identify Non-Automated Activities and Gateways

As previously mentioned, you would ideally like all of the phases of the pipeline to be automated.
In this ideal world, developers would check in code and release candidates would then flow through the pipeline, with each step automated and each gateway between phases automated. Eligible release candidates would be emitted at the end of the pipeline ready for deployment, and you would have complete confidence in every one.

For various reasons however, this is not always viable. For instance, common problems are a shortfall in the amount of automated testing, or business requirements for manual user acceptance, or beta testing. Even where high degrees of automated testing are in place, many businesses would likely include manual sign-offs before builds can flow through the pipeline and into production.

For this reason, our deployment pipeline definitely needs to acknowledge, model, and allow for humans and manual phases in the process:
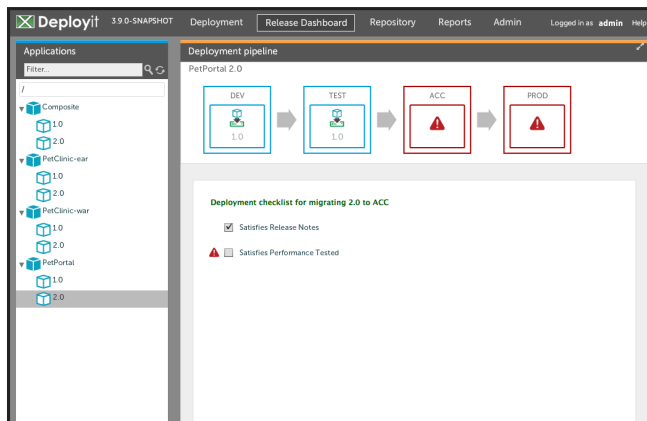
In some instances, you will find that the gateways between phases can also be automated. For instance, you might allow software to flow into a development- and performance-testing environment automatically if it passes automated testing within the continuous integration server, but you might want out exploratory test environment deploys to be controlled by QA staff as a manual and ideally self service step.

For both automated and manual gates, you will want to identify the criteria that make them pass. If gate criteria are not met, our system should prevent release candidates from progressing through the deployment pipeline.

Step 3 - Implement Your Pipeline
Once you've modeled the flow of your pipeline, you'll then have the fun task of actually implementing it.

Most of the implementation workfalls under the category of general release automation, as discussed in the above topic. If you can automate the main tasks around compilation, testing, and deployments, then you are in good shape to tie this together in the context of the deployment pipeline.

To manage your pipeline, you will need to make the choice between either building proprietary scripting or making the investment in off-the-shelf Application release management tooling to implement the processes.

Do not discount vendor tools, as these can potentially free up valuable developer and system administrator time that would otherwise be spent managing internal infrastructure and developing release automation glue code. Good software in this category will:

- Formalize the stages and the flows which your software goes through;
- Define the criteria that must be met for release candidates to move between stages in the pipeline;
- Allow you to parallelize stages of the pipeline where appropriate;
- Report and audit on deployments for management and operational purposes;
- Give you visibility into your pipeline, showing how builds have progressed and what specific changes are associated with each release candidate;
- Give your teams self-service facilities, for instance allowing operations staff to deploy to production and QA to bring a version into their test environment when they are ready. A permission model may be incorporated so that only certain authorized people have deployment permissions.

CONTINUOUS DELIVERY BEST PRACTICES

Once you've put the fundamentals in place and set up a deployment pipeline, you'll hopefully already begin to benefit from decreased cycle times and faster delivery.

Automation should be taking care of lots of the manual jobs, environments and deployments should be more consistent, and release candidates should be flowing between pipeline phases using automated gates and self-service tools.

Your software should almost always be in a production-ready state with release candidates coming out of the end of the pipeline much more frequently than the traditional delivery process. Each release candidate should be adding a relatively small batch of change.

Once you are at this stage, there is always more you can do to improve and move towards even faster delivery cycles while enhancing stability of your system.

A few of these best practices are listed below.

Implement Monitoring
Though everything we have discussed in this document describes a rigorous process which will help you to avoid releasing bugs into production, it's also important that you are alerted if something does go wrong in the system as a result of a deployment.

For instance, if your application starts throwing alerts or exceptions after a deployment, it's important that you are told straight away so that you can investigate and resolve.

Ideally this alert will be delivered via some dashboard or monitoring front-end, though an email, SMS, or something similar could work as a first step. You may also need to go a layer deeper than simply checking for errors in logs, and start monitoring the metrics that your application is pushing out. For instance, if your shopping cart completion rate drops by 20% after a release ,then this could indicate a more subtle but very serious error with the latest deployment. Open source tools such as StatsD and Graphite can help here. Again, these metrics should be pushed into your monitoring and alerting dashboards when possible.

There are a wealth of open source and hosted tools that can help you with intelligent monitoring of your applications. These are definitely worth evaluating as a fast and cost-effective means of supporting your continuous delivery project.

Best Practices:
Provide real time monitoring of your application, ideally via a visual dashboard.
Track key metrics regarding your applications usage, and alert if a deployment appears to negatively impact these.

Implement Rollback

Being able to quickly and reliably roll back changes applied to your production environment is the ultimate safety net.

If a bug slips through despite all of your automated and manual testing, the rollback will enable you to quickly move back to the previous working version of the software before too many users are impacted.

With the comfort of a good rollback process, you can then be even more ambitious in terms of automating deployment pipeline stages and opening up the gates between stages.

This confidence will result in a much faster pace of delivery, and you'll feel more confident in adding automation into the delivery process.

If the delivery pipeline is implemented well, you should essentially get this rollback capability for free. If version 9 of your software breaks, simply deploy version 8, which should still be there, all signed off and ready to re-deploy at the end of your pipeline.

Best Practices:
Provide a mechanism to quickly and repeatedly roll back any software changes to the system.
Build this into the pipeline process to reduce the need for developers to explicitly coding for rollbacks.
Test your rollback regularly as part of your pipeline to retain confidence in your process.

Extract Environment-specific Configuration

When moving through the deployment pipeline, it's important that you use the same binaries and artifacts right through the pipeline. If QA and UAT occur against different binaries, your testing is completely invalidated.

For this reason, you need the ability to push the same application binaries into arbitrary environments, and then also deploy environment-specific configuration separately.   This configuration should also be version controlled just like any other code, giving you much more repeatability and a better audit trail.

Quite often, the main stumbling block to doing this is having environment-specific configuration tied too tightly to the actual application binaries.

Extracting this environment-specific configuration into external properties files or other configuration sources gives us much more agility with regards to this.

Best Practices:
Use the same application binaries throughout your deployment pipeline. Avoid rebuilding from source or processing binaries in any way, even if you believe it is safe to do so.

Best Practices:
Extract environment-specific information into version controlled configuration that can be deployed separately from the main deployment artifacts.

Perform Canary Releases

A really useful technique for increasing stability of your production environment is the canary release.

This concept involves releasing the next version of your software into production, but only exposing it to a small percentage of your user base. Though the aim is never to introduce any bugs into our production environment, if you do, you would obviously prefer to insulate the bulk of the user base from any issues.

As you build confidence in the deployment, you can then increasingly expose more of the user base to it, until the previous version is completely out of scope.

Being able to canary release is a huge win with regards to continuous delivery, but can require significant work and architectural changes in the application.

Best Practices:
Deliver the capability to canary release, ideally while the production application is in use by users.

Capture Build Audit Information

Ideally, your continuous deployment pipeline should give you a very clear pattern of what specifically has changed about the software with each release candidate.   This has benefits all the way through the pipeline. Manual testing can specifically focus on the areas that have changed and you can move forward with more confidence if you know the exact scope of each deployment.

By integrating your continuous integration server with issue-tracking software such as Jira, you can develop highly-automated systems where release notes can be built and linked back to individual issues or defects. You should also be sure to capture all binaries that are released into environment for traceability and investigative reasons. Repository management tools such as Nexus can help here.

Best Practices:
Integrate with issue tracking or change management software to provide detailed audits of issues that are addressed with each release candidate.
Change-control all relevant code and archive all released binaries.

Implement Feature Flags

Feature flags are a facility that developers build into the software that gives them the ability to toggle specific features on or off with a high degree of granularity. This simple technique can add stability into the system through greater control of how new features are put into production use:
Good feature flags will be:

- Managed at runtime without a restart or user interruption.
- Well tested, in that you should make sure your tests cover all the combinations of features that you plan to use in production.
- Identifiable at runtime, allowing you to identify which features are active where, and how they correlate with usage of the system.

Best Practices:
Implement feature flags, with full understanding of implications for QA and production operations.

Use Cloud-based and Managed Infrastructure

Throughout this document there have been many mentions of cloud and various managed infrastructure providers. This is because they represent a fast and cost effective way of speeding up your continuous delivery efforts. Your core competency is in writing your own application code, rather than writing supporting release automation scripts.

Cloud and managed infrastructure as a service is particularly useful is in supporting teams that have variable requirements of their build and release infrastructure. For instance, you may find that you need to temporarily increase your capacity as you approach release.

The combination of cloud and automated infrastructure management are ideal for handling this variability in your CI requirements. For this reason, cloud-hosted services like CloudBees DEV@cloud (managed Jenkins in the cloud) are ideal candidates for outsourcing your Continuous Delivery infrastructure.

Best Practices:

Reduce infrastructure administration and management, and support variability in your infrastructure requirements by deploying onto the cloud or infrastructure as a service.

CONTINUOUS DELIVERY CHECKLIST

Fundamentals - Release Automation
- ✓ Automated Compilation and Packaging
- ✓ Automated Builds and Continuous Integration
- ✓ Automated Testing
- ✓ Automated Deployments
- ✓ Managed Infrastructure and Cloud
- ✓ Infrastructure As Code
- ✓ Automated Production Deployments

Implement A Deployment Pipeline
- ✓ Model Your Pipeline
- ✓ Identify Non-automated Activities and Gateways
- ✓ Implement Your Pipeline

Best Practices
- ✓ Implement Monitoring
- ✓ Implement Rollback
- ✓ Extract Environment-specific Configuration
- ✓ Perform Canary Releases
- ✓ Capture Audit Information
- ✓ Implement Feature Flags
- ✓ Use Cloud-based Infrastructure

Related Refcardz

There are a number of other Refcardz that provide more detail on release automation and the move towards Continuous Delivery:

Continuous Delivery Patterns:
http://refcardz.dzone.com/refcardz/continuous-delivery-patterns

Deployment Automation Patterns:
http://refcardz.dzone.com/refcardz/deployment-automation-patterns

Software Configuration Management Patterns:
http://refcardz.dzone.com/refcardz/software-configuration

Jenkins Continuous Integration:
http://refcardz.dzone.com/refcardz/jenkins-paas

Chef:
http://refcardz.dzone.com/refcardz/chef-open-source-tool-scalable

ABOUT THE AUTHORS
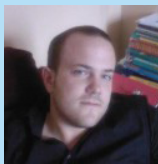
Benjamin Wootton (@benjaminwootton) is the Principle Consultant at Autumn Devops (autumndevops.com), a London, UK based consultancy specializing in DevOps and software release automation. He has over 10 years experience working at the intersection of agile Java software development and operations. He is the maintainer of the popular DevOps Friday (devopsfriday.com) newsletter.

Mark Prichard (@mqprichard) is Java PaaS Evangelist for CloudBees. He came to CloudBees after 13 years at BEA Systems and Oracle, where he was Product Manager for the WebLogic Platform.

Andrew Phillips is VP of Product Management for XebiaLabs. Andrew is a cloud, Continuous Delivery and automation expert and regularly contributes to key industry discussions of automated application delivery platforms.

RECOMMENDED BOOK
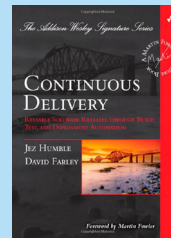
The authors introduce state-of-the-art techniques, including automated infrastructure management and data migration, and the use of virtualization. For each, they review key issues, identify best practices, and demonstrate how to mitigate risks. Whether you're a developer, systems administrator, tester, or manager, this book will help your organization move from idea to release faster than ever—so you can deliver value to your business rapidly and reliably.

BUY HERE

Browse our collection of over 150 Free Cheat Sheets
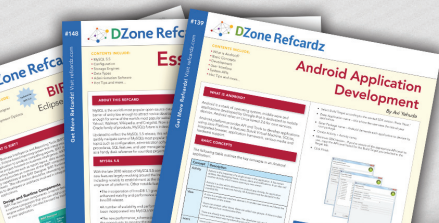
Free PDF

Upcoming Refcardz

C++
Cypher
Clean Code
Debugging Patterns

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. **""DZone is a developer's dream"**,—says PC Magazine.

ISBN-13: 978-1-936502-79-0
ISBN-10: 1-936502-79-8
50795

9 781936 502790

$7.95

Version 1.0