Continuous Delivery

By Paul M. Duvall

ABOUT CONTINUOUS DELIVERY

With Continuous Delivery (CD), teams continuously deliver new versions of software to production by decreasing the cycle time between an idea and usable software through the automation of the entire delivery system: build, deployment, test, and release. CD is enabled through the Deployment Pipeline, which encompasses a collection of patterns described in this Refcard.

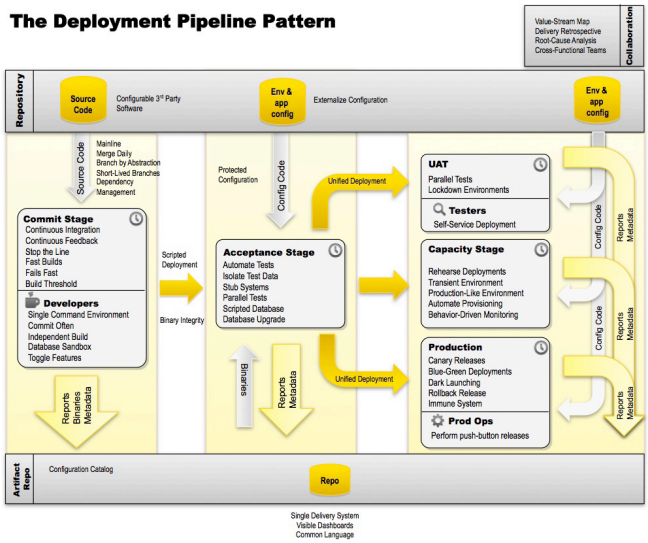
CD is concerned with "…how all the moving parts fit together: configuration management, automated testing, continuous integration and deployment, data management, environment management, and release management." *(Humble, et al.)*

THE DEPLOYMENT PIPELINE

The purpose of the deployment pipeline is threefold:

- **Visibility:** Makes every part of the process of building, deploying, testing, and releasing software visible to everybody involved, aiding collaboration
- **Feedback:** Improves feedback so that problems are identified and resolved as early in the process as possible
- **Continually Deploy:** Deploy and release any version of your software to any environment through a fully automated process" *(Humble, et al.)*

The Deployment Pipeline Pattern



BENEFITS

- **Empowering Teams:** Because the deployment pipeline is a pull system, testers, developers, operations, and others can self service the application version into the environment of their choice.
- **Reducing Errors:** Ensuring the correct version, configuration, database schema, etc. are applied the same way every time through automation.
- **Lowering Stress:** Through push-button releases to production and Rehearsing Deployments, a release becomes commonplace without the typical stress
- **Deployment Flexibility:** Instantiate a new environment or configuration by making a few changes to the automated delivery system.
- **Practice makes Perfect:** Through the deployment pipeline, the final deployment into production is being rehearsed every single time the software is deployed to any target environments. *(Humble, et al., 2011)*

CONFIGURATION MANAGEMENT

Configuration Management is "the process by which all artifacts relevant to your project, and the relationships between them, are stored, retrieved, uniquely identified, and modified". *Continuous Delivery (Humble, et al., 2011)*

Branch by Abstraction

Source: Paul Hammant and continuousdelivery.com

Pattern

Instead of using version-control branches, create an abstraction layer that handles both an old and new implementation. Remove the old implementation.

Anti-patterns

Branching using the version-control system leading to branch proliferation and difficult merging.

Configurable Third-party Software

Source: Continuous Delivery (Humble, et al., 2011)

Pattern
Evaluate and use third-party software that can be easily configured, deployed, and automated.

Anti-patterns
Purchasing software that makes it nearly impossible to configure. Software that forces teams to use the GUI interface only.

Configuration Catalog

Source: Continuous Delivery (Humble, et al., 2011)

Pattern
Maintain a catalog of all configuration options for each application and storage location and how to change these options. Automate through the build process, if possible.

Anti-patterns
Configuration options are not documented. The catalog of applications and other assets is "tribal knowledge".

Mainline

Source: Configuration Management Patterns (Berczuk, et al., 2002)

Pattern
Minimize merging and keep the number of active code lines manageable by developing on a mainline.

Anti-patterns
Multiple branches per project.

Merge Daily

Source: Continuous Delivery (Humble, et al., 2011)

Pattern
"Any changes from mainline must be merged onto every branch on a daily basis".

Anti-patterns
Merging every iteration once a week or less often than once a day.

Protected Configuration

Source: Continuous Integration Patterns (Duvall, 2010) and Continuous Delivery (Humble, et al., 2011)

Pattern
"Databases, directories, and registries are convenient places to store configuration since they can be accessed remotely."

Anti-patterns
Open text passwords on one person's machine.

Repository

Source: SCM Patterns (Berczuk, et al., 2002) and Continuous Delivery (Humble, et al., 2011)

Pattern
All source files are committed to version-control repository - executable code, configuration, host environment, and data.

Anti-patterns
Some files are checked in, others, such as environment configuration or data changes, are not. Binaries – that can be recreated through the build and deployment process – are checked in.

Short-Lived Branches

Source: Continuous Delivery (Humble, et al., 2011)

Pattern
Branches must be short-lived, ideally less than a few days, never more than an iteration.

Anti-patterns
Branches that last more than an iteration. Branches by product feature that live past a release.

Single Command Environment

Source: Continuous Delivery (Humble, et al., 2011)

Pattern
Check out the project's version-control repository and run a single command to build and deploy the application to any accessible environment, including the local development.

Anti-patterns
Requiring developer to define and configure environment variables. Require developer to install numerous tools in order for the build/deployment to work.

Single Delivery System

Source: Continuous Delivery (Humble, et al., 2011)

Pattern
"Configuration management of whole environments - the software, hardware, and infrastructure that an application depends upon; from operating systems to application servers, databases, and other commercial off-the-shelf (COTS) software". Should be able to associate any source file back to a single revision in version-control system.

Anti-patterns
Parts of system are not versioned. Inability to link hardware, network , data, and software changes together.

CONTINUOUS INTEGRATION

Build Threshold

Continuous Integration Patterns (Duvall, 2010)

Pattern
Fail a build when a project rule is violated – such as architectural breaches, slow tests, and coding standard violations.

Anti-patterns
Manual code reviews. Learning of code quality issues later in the development cycle.

Commit Often

Continuous Integration (Duvall, et al., 2007)

Pattern
Each team member should check in regularly to trunk - at least once a day, but preferrably after each task to trigger the Continuous Integration system.

Anti-patterns
"Source files stay checked out of a repository for long periods of time due to the amount of changes required for tasking."

Continuous Feedback

Continuous Integration (Duvall, et al., 2007)

Pattern
Send automated feedback from CI system to all Cross-Functional Team members.

Anti-patterns
"Teams choose not to send build status notifications to team members; thus, people aren't aware a build (or any part of the delivery system) has failed."

Continuous Integration

Continuous Integration (Duvall, et al., 2007)

Pattern
Building software with every change committed to a project's version control repository.

Anti-patterns
Scheduled builds, nightly builds, building periodically, building exclusively on developer's machines, not building at all.

Stop the Line

Source: Continuous Integration Patterns (Duvall, 2010) and Continuous Delivery (Humble, et al., 2011) and CI Cultural Maturity (Julius, 2009)

Pattern
Fix software delivery errors as soon as they occur: stop the line. No one checks in on a broken build as the fix becomes the highest priority.

Anti-patterns
Builds stay broken for long periods of time, thus preventing developers from checking out functioning code.

Independent Build

Source: Continuous Integration (Duvall, et al., 2007)

Pattern
Write build scripts that are decoupled from IDEs. These build scripts will be executed by a CI system so that software is built at every change.

Anti-patterns
Automated Build relies on IDE settings. Builds are unable to be run from the command line.

Visible Dashboards

Pattern
Provide large visible displays that aggregate information from your build system to provide high-quality feedback to the Cross-Functional Team in real time.

Anti-patterns
Email-only alerts or not publicizing the feedback to the entire team.

TESTING

Automate Tests

Pattern
Automate the verification and validation of software to include unit, component, capacity, functional, and deployment tests.

Anti-patterns
Manual testing of units, components, deployment, and other types of tests.

Unit: Automating tests that without any dependencies.

Component: Automating tests with dependencies to other components and heavyweight dependencies such as the database or file system.

Deployment: Automating the tests to verify the deployment and configuration were successful. Sometime referred to as a "smoke tests".

Functional: Automating the tests of the behavior of the software from the user's perspective.

Capacity: Automating load and performance testing in near-production conditions.

Isolate Test Data

Source: Continuous Delivery (Humble, et al., 2011)

Pattern
Use transactions for database-dependent tests (e.g., component tests) and roll back the transaction when done. Use a small subset of data to effectively test behavior.

Anti-patterns
Using a copy of production data for Commit Stage tests. Running tests against a shared database.

Parallel Tests

Source: Continuous Delivery (Humble, et al., 2011)

Pattern
Run multiple tests in parallel across hardware instances to decrease the time it takes to run tests.

Anti-patterns
Running tests on one machine or instance.

Stub Systems

Source: Continuous Delivery (Humble, et al., 2011)

Pattern
Use stubs to simulate external systems to reduce deployment complexity.

Anti-patterns
Manually installing and configuring interdependent systems for Commit Stage build and deployment.

DEPLOYMENT PIPELINE

Deployment Pipeline

Source: Continuous Delivery (Humble, et al., 2011)

Pattern
A deployment pipeline is an automated implementation of your application's build, deploy, test, and release process.

Anti-patterns
Deployments require human intervention (other than approval or clicking a button). Deployments are not production ready.

Value-Stream Map

Source: Lean Development (Poppendieck)

Pattern
Create a map illustrating the process between check in to the version-control to release to identify process bottlenecks.

Anti-patterns
Separate defined processes and views of the check in to release process.

BUILD AND DEPLOYMENT SCRIPTING

Common Language

Source: Continuous Delivery (Humble, et al., 2011)

Pattern
As a team, agree upon a common scripting language - such as Perl, Ruby or Python - so that any team member can apply changes to the Single Delivery System.

Anti-patterns
Each team uses a different language making it difficult for anyone to modify the delivery system reducing cross-functional team effectiveness.

Externalize Configuration

Continuous Integration Patterns (Duvall, 2010)

Pattern
Anything that changes between environments should be captured as configuration information. All variable values are externalized from the application configuration into build/deployment-time properties.

Anti-patterns
Hardcoding values inside the source code or per target environment.

Fail Fast

Pattern

Fail the build as soon as possible. Design scripts so that processes that commonly fail run first. These processes should be run as part of the Commit Stage.

Anti-patterns

Common build mistakes are no uncovered until later in the deployment process.

Fast Builds

Pattern

The Commit Build should provide feedback on common build problems as quickly as possible. Utilize "Cup of Coffee" metric as a general guideline on how quickly people should receive feedback on the most common errors.

Anti-patterns

Throwing everything into the commit stage process, such as running every type of automated static analysis tool or running load tests such that feedback is delayed.

Scripted Deployment

Source: Continuous Integration Patterns (Duvall, 2010)

Pattern

All deployment processes are written in a script, checked into the version-control system, and run as part of the Single Delivery System.

Anti-patterns

Deployment documentation is used instead of automation. Manual deployments or partially manual deployments. Using GUI to perform a deployment.

Unified Deployment

Source: Continuous Delivery (Humble, et al., 2011)

Pattern

"...Use the same script to deploy to every environment".

Anti-patterns

Different deployment script for each target environment or even for a specific machine. Manual configuration after deployment for each target environment.

DEPLOYING AND RELEASING APPLICATIONS

Binary Integrity

Continuous Integration Patterns (Duvall, 2010)

Pattern

Build your binaries once, while deploying the binaries to multiple target environments, as necessary.

Anti-patterns

Software is built in every stage of the deployment pipeline.

Canary Releases

Source: Continuous Delivery (Humble, et al., 2011)

Pattern

Release software to production for a small subset of users (e.g. 10%) to get feedback prior to a complete rollout.

Anti-patterns

Software is released to all users at once.

Blue-Green Deployments

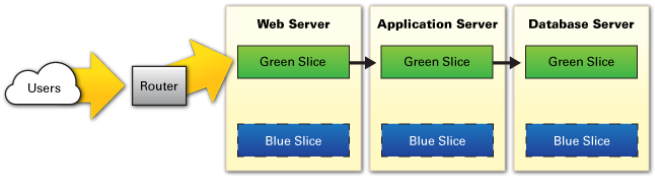
Source: Continuous Delivery (Humble, et al., 2011)

Pattern

Deploy software to a non-production environment (call it blue) while production continues to run. Once it's deployed and "warmed up", switch production (green) to non-production and blue to green simultaneously.

Anti-patterns

Production is taken down while the new release is applied to production instances(s).



Dark Launching

Source: Facebook Engineering Team

Pattern

Launch a new application or features when it will affect the least amount of users.

Anti-patterns

Software is deployed regardless of number of active users.

Rehearse Deployments

Source: Continuous Delivery (Humble, et al., 2011)

Pattern

Using the Unified Deployment pattern, continually practice deployments by running through the entire process via the Deployment Pipeline with each and every checkin.

Anti-patterns

Waiting until the end of the release cycle to test a realistic deployment.

Rollback Release

Continuous Integration Patterns (Duvall, 2010)

Pattern

Provide an automated single command rollback of changes after an unsuccessful deployment.

Anti-patterns

Manually undoing changes applied in a recent deployment. Shutting down production instances while changes are undone.

Self-Service Deployment

Source: Continuous Delivery (Humble, et al., 2011)

Pattern

Any Cross-Functional Team member can pick the version and environment and deploy the latest working software - giving everybody involved in the delivery process the ability to manage their own work.

Anti-patterns

Deployments are released to team are specified intervals by the "Build Team". Testing can only be performed in a shared state without isolation from others.

INFRASTRUCTURE AND ENVIRONMENTS

Automate Provisioning

Source: Continuous Delivery (Humble, et al., 2011)

Pattern

Automate the process of configuring your environment to include networks, external services, and infrastructure.

Anti-patterns

Configured instances are "works of art" requiring some or 100% manual steps to be performed to provision it.

Behavior-Driven Monitoring

Source: Continuous Delivery (Humble, et al., 2011)

Pattern
Automate tests to verify the behavior of the infrastructure. Continually run these tests to provide near real-time alerting.

Anti-patterns
No real-time alerting or monitoring. System configuration is written without tests.

Immune System

Source: IMVU

Pattern
Deploy software one machine instance at a time while conducting Behavior-Driven Monitoring. If an error is detected during the incremental deployment, a Rollback Release is initiated to revert changes.

Anti-patterns
Non-incremental deployments without monitoring.

Lockdown Environments

Source: Continuous Delivery (Humble, et al., 2011)

Pattern
"...Lock down the production environments to prevent unauthorized access not only from people outside your organization, but also from people within it - even operations staff."

Anti-patterns
The "Wild West": any authorized user can access machines and apply manual configuration changes, putting the environment in an unknown state.

Production-Like Environments

Source: Continuous Delivery (Humble, et al., 2011)

Pattern
Target environments should be as similar to production as possible.

Anti-patterns
Environments are "production like" weeks or days before a release. Environments are manually configured and controlled.

Transient Environments

Pattern
Utilizing the Automate Provisioning, Scripted Deployment, and Scripted Database patterns, any environment should be capable of terminating and launching at will.

Anti-patterns
Environments are fixed to "DEV, QA" and other.

DATA

Database Sandbox

Source: Refactoring Databases (Ambler, et al., 2011)

Pattern
Create a lightweight version of your database – using the Isolate Test Data pattern. Use this lightweight DML to populate local database sandboxes for each developer. Use this data in development environments to expedite test execution.

Anti-patterns
Shared database. Developers and testers are unable to make data changes without it potentially adversely affecting other team members immediately.

Database Upgrade

Source: Refactoring Databases (Ambler, et al., 2011)

Pattern
Use scripts to apply incremental changes in each target environment to a database schema and data.

Anti-patterns
Manually applying database and data changes in each target environment.

Scripted Database

Source: Refactoring Databases (Ambler, et al., 2011)

Pattern
Script all database actions as part of the build process.

Anti-patterns
Using data export/import to apply data changes. Manually applying schema and data changes to the database.

COMPONENTS AND DEPENDENCIES

Dependency Management

Source: Continuous Integration Patterns (Duvall, 2010)

Pattern
Centralize all dependent libraries to reduce bloat, classpath problems, and repetition of the same dependent libraries and transitive dependencies from project to project.

Anti-patterns
Multiple copies of the same binary dependencies in each and every project. Redefining the same information for each project. Classpath hell!

Toggle Features

Source: Martin Fowler

Pattern
Hide new functionality until it is complete. Deploy the software with the incomplete functionality but hide it from the user.

Anti-patterns
Waiting until a feature is fully complete before committing the source code.

COLLABORATION

Delivery Retrospective

Source: Continuous Delivery (Humble, et al., 2011)

Pattern
For each iteration, hold a retrospective meeting where everybody on the Cross-Functional Team discusses how to improve the delivery process for the next iteration.

Anti-patterns
Waiting until something goes wrong with a deployment for Dev and Ops to meet. Having Dev and Ops work separately.

Cross-Functional Teams

Source: Continuous Delivery (Humble, et al., 2011)

Pattern
Everybody is responsible for the delivery process. Any person on the Cross-Functional Team can modify any part of the delivery system.

Anti-patterns
Siloed teams: Development, Testing, and Operations have their own scripts, processes and are not part of the same team.

Amazon.com has an interesting take on this approach. They call it "You build it, you run it". Developers take the software they've written all the way to production.

Root-Cause Analysis

Source: Continuous Delivery (Humble, et al., 2011), Toyota Production System, Kaizen, and numerous other sources.

Pattern

Learn the root cause of a delivery problem by asking "why" of each answer and symptom until discovering the root cause.

Anti-patterns

Accepting the symptom as the root cause of the problem.

TOOLS

The following gives you an idea of the types of tools and some of the vendors that help enable effective Continuous Delivery. The Java, .NET, and Ruby platforms are represented.

Tool Type	Example Software Tools
Application Containers	JBoss, Tomcat, IIS, Mongrel
Build	Ant, AntContrib, NAnt, MSBuild, Buildr, Gant, Gradle, make, Maven, Rake
Continuous Integration	Bamboo, Jenkins, AntHill Pro, Go, TeamCity, TFS 2010
Cloud IaaS	AWS EC2, AWS S3 , Windows Azure
Cloud PaaS	Google App Engine, AWS Elastic Beanstalk, Heroku
Database	Hibernate, MySQL, Liquibase, Oracle, PostgreSQL, SQL Server, SimpleDB, SQL Azure, Ant, MongoDB
Database Change Management	dbdeploy, Liquibase
Data Center Configuration Automation	Capistrano, Cobbler, BMC Bladelogic, CFEngine, IBM Tivoli Provisioning Manager, Puppet, Chef, Bcfg2, AWS Cloud Formation, Windows Azure AppFabric, rPath
Dependency Management	Ivy, Archiva, Nexus, Artifactory, Bundler
Deployment Automation	Java Secure Channel, ControlTier, Altiris, Capistrano, Fabric, Func
Integrated Development Environment (IDE)	Eclipse, IDEA, Visual Studio
Issue Tracking	Greenhopper, JIRA
Passwords	PassPack, PasswordSafe
Protected Configuration	ESCAPE, ConfigGen
Provisioning	JEOS, BoxGrinder, CLIP, Eucalyptus, AppLogic
Reporting/Documentation	Doxygen, Grand, GraphViz, JavaDoc, NDoc, SchemaSpy, UmlGraph
Static Analysis	CheckStyle, Clover, Cobertura, FindBugs, FxCop, JavaNCSS, JDepend, PMD, Sonar, Simian
Systems Monitoring	CloudKick, Nagios, Zabbix, Zenoss
Testing	AntUnit, Cucumber, DbUnit, webrat, easyb, Fitnesse, JMeter, JUnit, NBehave, SoapUI, Selenium, RSpec, SauceLabs, Twist
Version-Control System	SVN/Subversion, git, Perforce

ABOUT THE AUTHOR

RECOMMENDED BOOK

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.

DZone, Inc.
140 Preston Executive Dr.
Suite 100
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-40-0
ISBN-10: 1-936502-40-2
50795
9 781936 502400
$7.95

Version 1.0