CONTENTS INCLUDE:
- The Deployment Pipeline
- Benefits
- Configuration Management
- Continuous Integration
- Testing
- Hot Tips and more...

Continuous Delivery
Patterns and Antipatterns in the Software Lifecycle
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.” (1)

THE DEPLOYMENT PIPELINE

- Visibility: All aspects of the delivery system - building, deploying, testing, and releasing – are visible to all team members promoting collaboration.

- Feedback: Team members learn of problems as soon as they occur so that issues are fixed as soon as possible.

- Continually Deploy: Through a fully automated process, you can deploy and release any version of the software to any environment. (1)

In the Deployment Pipeline diagram above, all of the patterns are shown in context. There are some patterns that span multiple stages of the pipeline, so I chose the stage where it’s most predominately used.

- Empowering Teams: Because the deployment pipeline is a pull system, testers, developers, operations, and others can self service the application version into an 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. (1)

The purpose of the deployment pipeline is threefold:

- Visibility: All aspects of the delivery system - building, deploying, testing, and releasing – are visible to all team members promoting collaboration.
# 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”. (1)

**Note:** Each pattern is cited with a number in parentheses that corresponds to the source in the References section.

## Configurable Third-Party Software (1)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Evaluate and use third-party software that can be easily configured, deployed, and automated.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Procuring software that cannot be externally configured. Software without an API or command-line interface that forces teams to use the GUI only.</td>
</tr>
</tbody>
</table>

## Configuration Catalog (1)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Maintain a catalog of all options for each application, how to change these options and storage locations for each application. Automatically create this catalog as part of the build process.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Configuration options are not documented. The catalog of applications and other assets is “tribal knowledge”.</td>
</tr>
</tbody>
</table>

## Mainline (3)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Minimize merging and keep the number of active code lines manageable by developing on a mainline.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Multiple branches per project.</td>
</tr>
</tbody>
</table>

## Merge Daily (1)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Changes committed to the mainline are applied to each branch on at least a daily basis.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Merging every iteration once a week or less often than once a day.</td>
</tr>
</tbody>
</table>

## Protected Configuration (5) , (1)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Store configuration information in secure remotely accessible locations such as a database, directory, or registry.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Open text passwords and/or single machine or share.</td>
</tr>
</tbody>
</table>

## Repository (3) , (1)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>All source files - executable code, configuration, host environment, and data - are committed to a version-control repository.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>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.</td>
</tr>
</tbody>
</table>

## Continuous Integration (CI)

### Build Threshold (5)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Fail a build when a project rule is violated – such as architectural breaches, slow tests, and coding standard violations.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Manual code reviews. Learning of code quality issues later in the development cycle.</td>
</tr>
</tbody>
</table>

### Commit Often (6)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Each team member checks in regularly to trunk - at least once a day but preferably after each task to trigger the CI system.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Source files are committed less frequently than daily due to the number of changes from the developer.</td>
</tr>
</tbody>
</table>

### Continuous Feedback (6)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Send automated feedback from CI system to all Cross-Functional Team members.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Notifications are not sent; notifications are ignored; CI system spams everyone with information they cannot use.</td>
</tr>
</tbody>
</table>

### Continuous Integration (6)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Building and testing software with every change committed to a project’s version control repository.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Scheduled builds, nightly builds, building periodically, building exclusively on developer’s machines, not building at all.</td>
</tr>
</tbody>
</table>
### Stop the Line (5), (1), (4), (12)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Builds stay broken for long periods of time, thus preventing developers from checking out functioning code.</td>
</tr>
</tbody>
</table>

### Independent Build (6)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Write build scripts that are decoupled from IDEs. These build scripts are executed by a CI system so that software is built at every change.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Automated build relies on IDE settings. Builds are unable to be run from the command line.</td>
</tr>
</tbody>
</table>

### Visible Dashboards

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Provide large visible displays that aggregate information from your delivery system to provide high-quality feedback to the Cross-Functional Team in real time.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Email-only alerts or not publicizing the feedback to the entire team.</td>
</tr>
</tbody>
</table>

### TESTING

#### Automate Tests

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Automate the verification and validation of software to include unit, component, capacity, functional, and deployment tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Manual testing of units, components, deployment, and other types of tests.</td>
</tr>
</tbody>
</table>

- **Unit** - Automating tests without any dependencies.
- **Component** - Automating tests with dependencies to other components and heavyweight dependencies such as the database or file system.
- **Deployment** - Automating tests to verify the deployment and configuration were successful. Sometimes referred to as a “smoke tests”.
- **Functional** - Automating tests to verify the behavior of the software from a user’s perspective.
- **Capacity** - Automating load and performance testing in near-production conditions.

#### Isolate Test Data (1)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Using a copy of production data for Commit Stage tests. Running tests against a shared database.</td>
</tr>
</tbody>
</table>

### Parallel Tests (1)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Run multiple tests in parallel across hardware instances to decrease the time in running tests.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Running tests on one machine or instance. Running dependent tests that cannot be run in parallel.</td>
</tr>
</tbody>
</table>

### Stub Systems (1)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Use stubs to simulate external systems to reduce deployment complexity.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Manually installing and configuring interdependent systems for Commit Stage build and deployment.</td>
</tr>
</tbody>
</table>

### DEPLOYMENT PIPELINE

#### Deployment Pipeline (1)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>A deployment pipeline is an automated implementation of your application’s build, deploy, test, and release process.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Deployments require human intervention (other than approval or clicking a button). Deployments are not production ready.</td>
</tr>
</tbody>
</table>

### Value-Stream Map (4)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Create a map illustrating the process from check in to the version-control system to the software release to identify process bottlenecks.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Separately defined processes and views of the checkin to release process.</td>
</tr>
</tbody>
</table>

### BUILD AND DEPLOYMENT SCRIPTING

#### Dependency Management (5)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Centralize all dependent libraries to reduce bloat, classpath problems, and repetition of the same dependent libraries and transitive dependencies from project to project.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Multiple copies of the same binary dependencies in each and every project. Redefining the same information for each project. Classpath hell!</td>
</tr>
</tbody>
</table>

#### Common Language (1)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Each team uses a different language making it difficult for anyone to modify the delivery system reducing cross-functional team effectiveness.</td>
</tr>
</tbody>
</table>

#### Externalize Configuration (5)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Changes between environments are captured as configuration information. All variable values are externalized from the application configuration into build/deployment-time properties.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-patterns</td>
<td>Hardcoding values inside the source code or per target environment.</td>
</tr>
</tbody>
</table>
## Continuous Delivery: Patterns and Antipatterns in the Software Life Cycle

### Fail Fast (6)

**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 not uncovered until late in the deployment process.

### Fast Builds (6)

**Pattern**
The Commit Build provides feedback on common build problems as quickly as possible - usually in under 10 minutes.

**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 (5)

**Pattern**
All deployment processes are written in a script, checked in to 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 (5)

**Pattern**
The same deployment script is used for each deployment. The Protected Configuration - per environment - is variable but managed.

**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 (5)

**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 Release

**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.

### INFRASTRUCTURE AND ENVIRONMENTS

#### Automate Provisioning (1)

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

**Anti-patterns**
Configured instances are “works of art” requiring team members to perform partially or fully manual steps to provision them.

#### Behavior-Driven Monitoring (1)

**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.

## Blue-Green Deployments (1)

**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 instance(s).

## Dark Launching (11)

**Pattern**
Launch a new application or features when it affects the least amount of users.

**Anti-patterns**
Software is deployed regardless of number of active users.

## Rollback Release (5)

**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 (1)

**Pattern**
Any Cross-Functional Team member selects the version and environment to deploy the latest working software.

**Anti-patterns**
Deployments released to team are at specified intervals by the “Build Team”. Testing can only be performed in a shared state without isolation from others.
### Immune System (9)

**Pattern**
Deploy software one 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 (1)

**Pattern**
Lock down shared environments from unauthorized external and internal usage, including operations staff. All changes are versioned and applied through automation.

**Anti-patterns**
The “Wild West”: any authorized user can access shared environments and apply manual configuration changes, putting the environment in an unknown state leading to deployment errors.

### Production-Like Environments (1)

**Pattern**
Target environments are as similar to production as possible.

**Anti-patterns**
Environments are “production like” only 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” or other pre-determined environments.

### Database Sandbox (7)

**Pattern**
Create a lightweight version of your database – using the Isolate Test Data pattern. Each developer uses this lightweight DML to populate his local database sandboxes 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.

### Decouple Database (1)

**Pattern**
Ensure your application is backward and forward compatible with your database so you can deploy each independently.

**Anti-patterns**
Application code data are not capable of being deployed separately.

### Database Upgrade (7)

**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 (7)

**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.

### INCREMENTAL DEVELOPMENT

#### Branch by Abstraction (2)

**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. Feature branching.

#### Toggle Features (10)

**Pattern**
Deploy new features or services to production but limit access dynamically for testing purposes.

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

### COLLABORATION

#### Delivery Retrospective (1)

**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 an error occurs during a deployment for Dev and Ops to collaborate. Having Dev and Ops work separately.

#### Cross-Functional Teams (1)

**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 and 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 (1)

**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

This is meant to be an illustrative list, not an exhaustive list, to give you an idea of the types of tools and some of the vendors that help to enable effective Continuous Delivery. The Java, .NET and Ruby platforms are represented. The tools that span categories have been...
Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation

The book has two themes: automation and collaboration. Delivering software of any complexity involves people with a bunch of skills: testers, developers, and sysadmin / operations personnel.

The last group of people often gets left out of the process of delivering software until the end, and even testers are often not as heavily involved in the development process as they should be.