The Addison-Wesley Signature Series

A Martin Fowler Signature Book

Continuous Delivery

Reliable Software Releases through Build, Test, and Deployment Automation

Jez Humble
David Farley

Foreword by Martin Fowler

Praise for *Continuous Delivery*

"If you need to deploy software more frequently, this book is for you. Applying it will help you reduce risk, eliminate tedious work, and increase confidence. I'll be using the principles and practices here on all my current projects."

—Kent Beck, Three Rivers Institute

"Whether or not your software development team already understands that continuous integration is every bit as necessary as source code control, this is required reading. This book is unique in tying the whole development and delivery process together, providing a philosophy and principles, not just techniques and tools. The authors make topics from test automation to automated deployment accessible to a wide audience. Everyone on a development team, including programmers, testers, system administrators, DBAs, and managers, needs to read this book."

—Lisa Crispin, co-author of Agile Testing

"For many organizations Continuous Delivery isn't just a deployment methodology, it's critical to doing business. This book shows you how to make Continuous Delivery an effective reality in your environment."

—James Turnbull, author of Pulling Strings with Puppet

"A clear, precise, well-written book that gives readers an idea of what to expect for the release process. The authors give a step-by-step account of expectations and hurdles for software deployment. This book is a necessity for any software engineer's library."

—Leyna Cotran, Institute for Software Research, University of California, Irvine

"Humble and Farley illustrates what makes fast-growing web applications successful. Continuous deployment and delivery has gone from controversial to commonplace and this book covers it excellently. It's truly the intersection of development and operations on many levels, and these guys nailed it."

—John Allspaw, VP Technical Operations, Etsy.com and author of The Art of Capacity Planning *and* Web Operations

"If you are in the business of building and delivering a software-based service, you would be well served to internalize the concepts that are so clearly explained in Continuous Delivery. But going beyond just the concepts, Humble and Farley provide an excellent playbook for rapidly and reliably delivering change."

—Damon Edwards, President of DTO Solutions and co-editor of dev2ops.org

"I believe that anyone who deals with software releases would be able to pick up this book, go to any chapter and quickly get valuable information; or read the book from cover to cover and be able to streamline their build and deploy process in a way that makes sense for their organization. In my opinion, this is an essential handbook for building, deploying, testing, and releasing software."

—Sarah Edrie, Director of Quality Engineering, Harvard Business School

"Continuous Delivery is the logical next step after Continuous Integration for any modern software team. This book takes the admittedly ambitious goal of constantly delivering valuable software to customers, and makes it achievable through a set of clear, effective principles and practices."

—Rob Sanheim, Principal at Relevance, Inc.

Why Binaries Should Not Be Environment-Specific

We consider it a very bad practice to create binary files intended to run in a single environment. This approach, while common, has several serious drawbacks that compromise the overall ease of deployment, flexibility, and maintainability of the system. Some tools even encourage this approach.

When build systems are organized in this way, they usually become very complex very quickly, spawning lots of special-case hacks to cope with the differences and the vagaries of various deployment environments. On one project that we worked on, the build system was so complex that it took a full-time team of five people to maintain it. Eventually, we relieved them of this unpopular job by reorganizing the build and separating the environment-specific configuration from the environment-agnostic binaries.

Such build systems make unnecessarily complex what should be trivial tasks, such as adding a new server to a cluster. This, in turn, forces us into fragile, expensive release processes. If your build creates binaries that only run on specific machines, start planning how to restructure them now!

This brings us neatly to the next practice.

Deploy the Same Way to Every Environment

It is essential to use the same process to deploy to every environment—whether a developer or analyst's workstation, a testing environment, or production—in order to ensure that the build and deployment process is tested effectively. Developers deploy all the time; testers and analysts, less often; and usually, you will deploy to production fairly infrequently. But this frequency of deployment is the inverse of the risk associated with each environment. The environment you deploy to least frequently (production) is the most important. Only after you have tested the deployment process hundreds of times on many environments can you eliminate the deployment script as a source of error.

Every environment is different in some way. If nothing else, it will have a unique IP address, but often there are other differences: operating system and middleware configuration settings, the location of databases and external services, and other configuration information that needs to be set at deployment time. This does not mean you should use a different deployment script for every environment. Instead, keep the settings that are unique for each environment separate. One way to do this is to use properties files to hold configuration information. You can have a separate properties file for each environment. These files should be checked in to version control, and the correct one selected either by looking at the hostname of the local server, or (in a multimachine environment) through the use of an environment variable supplied to the deployment script. Some other ways to supply deploy-time configuration include keeping it in a directory

service (like LDAP or ActiveDirectory) or storing it in a database and accessing it through an application like ESCAPE [apvrEr]. There is more on managing software configuration in the "Managing Software Configuration" section on page 39.

It's important to use the same deploy-time configuration mechanism for each of your applications. This is especially true in a large company, or where many heterogeneous technologies are in play. Generally, we're against handing down edicts from on high—but we've seen too many organizations where it was impossibly arduous to work out, for a given application in a given environment, what configuration was actually supplied at deployment time. We know places where you have to email separate teams on separate continents to piece together this information. This becomes an enormous barrier to efficiency when you're trying to work out the root cause of some bug—and when you add together the delays it introduces into your value stream, it is incredibly costly.

It should be possible to consult one single source (a version control repository, a directory service, or a database) to find configuration settings for all your applications in all of your environments.

If you work in a company where production environments are managed by a team different from the team responsible for development and testing environments, both teams will need to work together to make sure the automated deployment process works effectively across all environments, including development environments. Using the same script to deploy to production that you use to deploy to development environments is a fantastic way to prevent the "it works on my machine" syndrome [c29ETR]. It also means that when you come to release, you will have tested your deployment process hundreds of times by deploying to all of your other environments. This is one of the best ways we know to mitigate the risk of releasing software.

We've assumed that you have an automated process for deploying your application—but, of course, many organizations still deploy manually. If you have a manual deployment process, you should start by ensuring that the process is the same for every environment and then begin to automate it bit by bit, with the goal of having it fully scripted. Ultimately, you should only need to specify the target environment and the version of the application to initiate a successful deployment. An automated, standardized deployment process will have a huge positive effect on your ability to release your application repeatably and reliably, and ensure that the process is completely documented and audited. We cover automating deployment in detail in the following chapter.

This principle is really another application of the rule that you should separate what changes from what doesn't. If your deployment script is different for different environments, you have no way of knowing that what you're testing will actually work when you go live. Instead, if you use the same process to deploy everywhere, when a deployment doesn't work to a particular environment you can narrow it down to one of three causes:

- A setting in your application's environment-specific configuration file
- A problem with your infrastructure or one of the services on which your application depends
- The configuration of your environment

Establishing which of these is the underlying cause is the subject of the next two practices.

Smoke-Test Your Deployments

When you deploy your application, you should have an automated script that does a smoke test to make sure that it is up and running. This could be as simple as launching the application and checking to make sure that the main screen comes up with the expected content. Your smoke test should also check that any services your application depends on are up and running—such as a database, messaging bus, or external service.

The smoke test, or deployment test, is probably the most important test to write once you have a unit test suite up and running—indeed, it's arguably even more important. It gives you the confidence that your application actually runs. If it doesn't run, your smoke test should be able to give you some basic diagnostics as to whether your application is down because something it depends on is not working.

Deploy into a Copy of Production

The other main problem many teams experience going live is that their production environment is significantly different from their testing and development environments. To get a good level of confidence that going live will actually work, you need to do your testing and continuous integration on environments that are as similar as possible to your production environment.

Ideally, if your production environment is simple or you have a sufficiently large budget, you can have exact copies of production to run your manual and automated tests on. Making sure that your environments are the same requires a certain amount of discipline to apply good configuration management practices. You need to ensure that:

- Your infrastructure, such as network topology and firewall configuration, is the same.
- Your operating system configuration, including patches, is the same.
- Your application stack is the same.
- Your application's data is in a known, valid state. Migrating data when performing upgrades can be a major source of pain in deployments. We deal more with this topic in Chapter 12, "Managing Data."

You can use such practices as disk imaging and virtualization, and tools like Puppet and InstallShield along with a version control repository, to manage your environments' configuration. We discuss this in detail in Chapter 11, "Managing Infrastructure and Environments."

Each Change Should Propagate through the Pipeline Instantly

Before continuous integration was introduced, many projects ran various parts of their process off a schedule—for example, builds might run hourly, acceptance tests nightly, and capacity tests over the weekend. The deployment pipeline takes a different approach: The first stage should be triggered upon every check-in, and each stage should trigger the next one immediately upon successful completion. Of course this is not always possible when developers (especially on large teams) are checking in very frequently, given that the stages in your process can take a not insignificant amount of time. The problem is shown in Figure 5.6.

In this example, somebody checks a change into version control, creating version 1. This, in turn, triggers the first stage in the pipeline (build and unit tests). This passes, and triggers the second stage: the automated acceptance tests. Somebody then checks in another change, creating version 2. This triggers the build and unit tests again. However, even though these have passed, they cannot trigger a new instance of the automated acceptance tests, since they are already running. In the meantime, two more check-ins have occurred in quick succession. However, the CI system should not attempt to build both of them—if it followed that rule, and developers continued to check in at the same rate, the builds would get further and further behind what the developers are currently doing.

Instead, once an instance of the build and unit tests has finished, the CI system checks to see if new changes are available, and if so, builds off the most recent set available—in this case, version 4. Suppose this breaks the build and unit tests stage. The build system doesn't know which commit, 3 or 4, caused the stage to break, but it is usually simple for the developers to work this out for themselves. Some CI systems will let you run specified versions out of order, in which case the developers could trigger the first stage off revision 3 to see if it passes or fails, and thus whether it was commit 3 or 4 that broke the build. Either way, the development team checks in version 5, which fixes the problem.

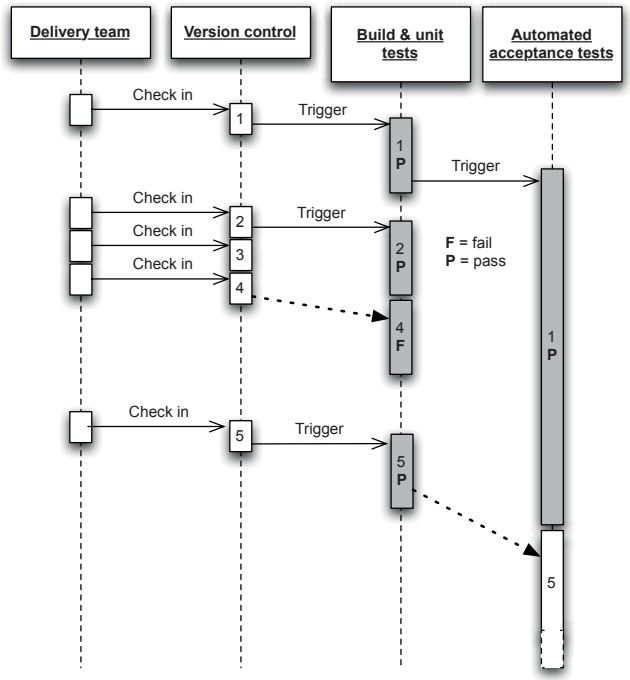

Figure 5.6 *Scheduling stages in a pipeline*

When the acceptance tests finally finish, the CI system's scheduler notices that new changes are available, and triggers a new run of the acceptance tests against version 5.

This intelligent scheduling is crucial to implementing a deployment pipeline. Make sure your CI server supports this kind of scheduling workflow—many do—and ensure that changes propagate immediately so that you don't have to run stages off a fixed schedule.

This only applies to stages that are fully automated, such as those containing automated tests. The later stages in the pipeline that perform deployments to manual testing environments need to be activated on demand, which we describe in a later section in this chapter.

If Any Part of the Pipeline Fails, Stop the Line

As we said in the "Implementing Continuous Integration" section on page 56, the most important step in achieving the goals of this book—rapid, repeatable, reliable releases—is for your team to accept that every time they check code into version control, it will successfully build and pass every test. This applies to the

entire deployment pipeline. If a deployment to an environment fails, the whole team owns that failure. They should stop and fix it before doing anything else.

The Commit Stage

A new instance of your deployment pipeline is created upon every check-in and, if the first stage passes, results in the creation of a release candidate. The aim of the first stage in the pipeline is to eliminate builds that are unfit for production and signal the team that the application is broken as quickly as possible. We want to expend a minimum of time and effort on a version of the application that is obviously broken. So, when a developer commits a change to the version control system, we want to evaluate the latest version of the application quickly. The developer who checked in then waits for the results before moving on to the next task.

There are a few things we want to do as part of our commit stage. Typically, these tasks are run as a set of jobs on a build grid (a facility provided by most CI servers) so the stage completes in a reasonable length of time. The commit stage should ideally take less than five minutes to run, and certainly no more than ten minutes. The commit stage typically includes the following steps:

- Compile the code (if necessary).
- Run a set of commit tests.
- Create binaries for use by later stages.
- Perform analysis of the code to check its health.
- Prepare artifacts, such as test databases, for use by later stages.

The first step is to compile the latest version of the source code and notify the developers who committed changes since the last successful check-in if there is an error in compilation. If this step fails, we can fail the commit stage immediately and eliminate this instance of the pipeline from further consideration.

Next, a suite of tests is run, optimized to execute very quickly. We refer to this suite of tests as commit stage tests rather than unit tests because, although the vast majority of them are indeed unit tests, it is useful to include a small selection of tests of other types at this stage in order to get a higher level of confidence that the application is really working if the commit stage passes. These are the same tests that developers run before they check in their code (or, if they have the facility to do so, through a pretested commit on the build grid).

Begin the design of your commit test suite by running all unit tests. Later, as you learn more about what types of failure are common in acceptance test runs and other later stages in the pipeline, you should add specific tests to your commit test suite to try and find them early on. This is an ongoing process optimization