

O'REILLY®

Continuous Delivery with Windows and .NET



Matthew Skelton
& Chris O'Dell

4 Easy Ways to Stay Ahead of the Game

The world of web ops and performance is constantly changing. Here's how you can keep up:

- 1 **Download free reports** on the current and trending state of web operations, dev ops, business, mobile, and web performance. http://oreil.ly/free_resources
- 2 **Watch free videos and webcasts** from some of the best minds in the field—watch what you like, when you like, where you like. http://oreil.ly/free_resources
- 3 **Subscribe** to the weekly O'Reilly Web Ops and Performance newsletter. <http://oreil.ly/getnews>
- 4 **Attend the O'Reilly Velocity Conference**, the must-attend gathering for web operations and performance professionals, with events in California, New York, Europe, and China. <http://velocityconf.com>

For more information and additional Web Ops and Performance resources, visit http://oreil.ly/Web_Ops.

Continuous Delivery with Windows and .NET

Matthew Skelton and Chris O'Dell

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Continuous Delivery with Windows and .NET

by Matthew Skelton and Chris O'Dell

Copyright © 2016 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Brian Anderson

Production Editor: Kristen Brown

Copyeditor: Lindsay Gamble

Proofreader: Jasmine Kwityn

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

February 2016: First Edition

Revision History for the First Edition

2016-01-25: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Continuous Delivery with Windows and .NET*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-94107-2

[LSI]

Table of Contents

Foreword.....	ix
Preface.....	xi
1. Introduction to Continuous Delivery.....	1
What Continuous Delivery Is Not	1
The Importance of Automation for Continuous Delivery	3
Why Is Continuous Delivery Needed?	4
Why Windows Needs Special Treatment	4
Terminology Used in This Book	5
2. Version Control.....	7
Key Version Control Practices	7
Version Control Technologies	8
Branching Options	11
Use NuGet for Dependencies	13
Summary	14
3. Continuous Integration.....	15
CI Servers for Windows and .NET	15
Build Automation	22
Integrating CI with Version Control and Ticket Tracking	25
Patterns for CI Across Multiple Teams	25
Architecture Changes for Better CI	26
Summary	26
4. Deployment Pipelines.....	27

Mapping Out a Deployment Pipeline	27
Tools for Deployment Pipelines	28
Deployment Techniques	32
Automated Testing of Database Changes	35
Summary	38
5. Monitoring, Metrics, and APM.	39
Performance Counters Are Insufficient	39
Record Application Metrics	39
APM Tools Can Complement Monitoring	41
Aggregate Application and Windows Event Logs from All Machines	43
Summary	46
6. Infrastructure Automation.	47
Shared Versus Dedicated Infrastructure	48
Using a Test-First Approach to Infrastructure	49
Patching and OS Updates	52
Summary	53
7. The Tricky Bits of Continuous Delivery.	55
Organizational Changes	55
Architectural Changes (SOA/Microservices)	57
Operational Features	59
Summary	60
A. Bibliography.	61
B. Case Studies.	63

Foreword

Continuous Delivery is widely seen as “state of the art” in today’s software development process. There are good reasons for this. Continuous Delivery is grounded in a pragmatic, empirical approach to software development. At its simplest, Continuous Delivery is about optimizing the whole software development process—from having an idea to getting that idea into the hands of our users, in the form of high-quality, working software, as quickly and efficiently as we can. Then we can figure out what our users make of our ideas.

We actively seek feedback loops like this within the development process, and look to optimize them. We encourage the adoption of a genuinely experimental approach to every aspect of software development. In fact, I think that over the past few years, we are seeing signs of the software development process maturing.

I think that what we are seeing is the recognition and establishment of what software engineering should really look like: high quality, iterative, experimental, empirical—in fact, grounded in the scientific method. I am probably biased, but I believe that Continuous Delivery represents an important step forward in the software development process. It works. It makes the organizations that practice it more efficient at creating high-quality software that delights their users.

You will notice that my description did not mention technology at all. That is because the ideas of Continuous Delivery are not tied to any specific technology. You can practice this approach to development for any software tech. However, if our aim is to “create a repeatable, reliable process for software delivery,” then automation plays an important part. After all, human beings are not really very good at being repeatable and reliable! So technology, though not the most important aspect, plays a vital role. On this technology front, the picture across the software industry is complex. There are at least two factions: the world of Windows and .NET, and the Unix-based tribes of the “Open Stack.”

I think it fair to say that some of the initial innovation in the Continuous Delivery space came from the Open Stack community—unit test frameworks, build management systems, sophisticated version

control and dependency management systems, and more recently support for ideas like “infrastructure as code” and “deployment pipelines.”

It has never been as simple as “Open Stack has the best tools and Windows plays catch-up.” The first publicly available Continuous Integration (build management) system was CruiseControl, followed quite quickly by a “port” to Windows called CruiseControl.NET, which was significantly better than the original. So over time, both worlds have developed their own collections of technology to support Continuous Delivery and its associated practices. Because many of the people that started the conversation on this new approach came from the Open Stack world (myself included), the language that we used and the tools that we described were not always descriptions or technologies that would resonate with someone from the Windows and .NET world.

It is great to see Matthew and Chris redress the balance. This book is much more than a simple description of Windows native tooling for Continuous Delivery, though that information is provided here. This book explores the ideas of Continuous Delivery and talks about the broad concepts and philosophy, as well as describing some of the specific tooling that works in a Windows-native way. In addition, it also describes some of the tools that are stack-neutral and available to any software developer whatever their toolset.

Chris and Matthew bring real-world experience to their writing, and back up their descriptions with firsthand experience of the technologies described and case studies from many different organizations.

If you are developing software of any kind in the Windows environment, this book is for you. Continuous Delivery is too important an idea to miss the boat. Enjoy the book!

—*Dave Farley*
Independent consultant and
coauthor of
Continuous Delivery

Preface

By re-architecting for Continuous Delivery, and using tools like Chef and GoCD in combination with Windows and .NET, we were able to move from infrequent, difficult code deployments to weekly, daily, and even hourly deployments, whilst improving our availability and mean time to recovery.

—John Esser, Ancestry.com

Continuous Delivery is a well-defined set of practices and approaches to releasing software in a reliable way. At the heart of Continuous Delivery is the automation of software builds, software testing, and software deployments, including the automated configuration of devices and environments for testing and runtime.

Organizations increasingly rely on software systems to enable and power their services, sales, or operations (or all three), and so frequent, reliable, repeatable, and rapid releases are essential in order to keep pace with demands for change. The practices that form Continuous Delivery are no longer optional for many organizations, but rather central to their very survival.

Today, all core software from Microsoft is PowerShell-scriptable, and teams working with Windows and .NET in 2016 and beyond are able to use a rich set of PowerShell and HTTP REST APIs in software packages and products from Microsoft, third-party vendors, and the open source community. This ability to automate Windows and .NET is a huge enabler for Continuous Delivery.

Many of the technologies and approaches for Continuous Delivery are essentially identical across different operating systems, but some things need special treatment for Windows and .NET. When Jez Humble and Dave Farley published their groundbreaking book,

Continuous Delivery [HumbleFarley], in 2010, many of the tools described were either nonexistent on the Windows platform or did not support the rich automation capabilities they do now. Our book acts as an addendum to Jez and Dave's book to encourage many more teams working with Windows and .NET to adopt Continuous Delivery.

Who Should Read This Book

If you build, release, or operate software systems built on .NET or Windows, then this book is for you. Architects, product managers, and CxOs in your organization should read **Chapter 7, *The Tricky Bits of Continuous Delivery***, and start planning some significant changes.

Product Development Best Practices

While good practices for product development with software, such as User Stories, are beyond the scope of this book, we refer to them occasionally. See the excellent book *Implementing Lean Software Development* [Poppendieck] for more details.

The Structure of the Book

A successful adoption of Continuous Delivery is built upon solid foundations of technical excellence, which we describe here:

Version control, branching, and merging (Chapter 2)

Everything in Continuous Delivery flows from sound version control.

Continuous Integration done well (Chapter 3)

Without CI for application and infrastructure code, Continuous Delivery is not possible.

Deployment pipelines (Chapter 4)

These are for deployment coordination, sharing information, and continuous improvement.

Infrastructure automation (Chapter 6)

Well-tested, well-structured code that defines servers and environments is needed.

The tricky bits (Chapter 7)

These things are as important as the more technical bits, but often get forgotten.

Many teams working with Windows/.NET have been slow to adopt modern, scriptable tools like NuGet and Chocolatey (package management), DSC/Chef/Puppet/Ansible (configuration management), and Vagrant/Packer/ServerSpec (server testing). We have tried to provide a flavor of the kinds of tools you should expect to use, the challenges you might face, and enough understanding of the principles to go and explore the tools and techniques for yourself.

In summary, enjoy the automation that modern Windows and .NET tools and software offer, but don't forget the social, architectural, and operational challenges that Continuous Delivery requires. We assure you that the changes will be worth it in the end!

NOTE

In this book, we use C# (so project files are *.csproj*) but most of the advice applies equally well to other .NET languages like VB.NET or F#.

We hope you find this book useful. We'd love to receive your feedback—send us an email at book@cdwithwindows.net or leave a comment on the book's website: <http://cdwithwindows.net/>.

Acknowledgments

We'd like to thank several people for their comments and support: first, all the people and organizations featured in the case study examples (Andy Lole at Carnect, Paul Shannon at 7digital, Steve Elliott at LateRooms.com, Peter Mounce at JUST EAT, Owain Perry at JustGiving, and John Esser and Russ Barnet at Ancestry.com). Second, we'd like to thank our technical reviewers for their insights (Josef Sin, John Adams, Colin Beales, and Giles Davies from the Visual Studio UK team, and Kundana Palagiri and colleagues from the Microsoft Azure team). Third, we're grateful to colleagues and friends who made suggestions, including Matt Richardson, Rob Thatcher, João Lebre, and Suzy Beck. Finally, we'd like to thank Dave Farley for writing the Foreword, and the editorial team at O'Reilly (including Brian Anderson and Kristen Brown) for asking us to write this book in the first place and making it ready for publication.

Introduction to Continuous Delivery

Continuous Delivery is one of the most valuable approaches to software development to emerge in recent years. At its core, Continuous Delivery is a set of practices and disciplines that enable organizations to reach and maintain a high-speed, predictable, steady, and safe stream of valuable software changes irrespective of the kind of software being developed. Continuous Delivery works not just for web-based software, but also mobile apps, on-premise hosted desktop software, device firmware, and so on.

In 2010, Jez Humble and Dave Farley wrote the book *Continuous Delivery* [HumbleFarley] based on their experiences building and releasing software for clients around the world. Their book is a hugely valuable collection of techniques, advice, and suggestions for software delivery and provides the de facto definition of Continuous Delivery.

What Continuous Delivery Is Not

Many people **confuse Continuous Delivery with Continuous Deployment**, but the two are quite different in purpose and execution. As seen in **Figure 1-1**, Continuous Delivery aims to enable regular, rapid, reliable software releases through a set of sound practices, giving the people who “own” the software product the power to decide when to release changes. Continuous Delivery is a so-called pull-based approach, because software is “pulled” through the deliv-

ery mechanism when needed, and applies to all kinds of software (web, desktop, embedded, etc.).

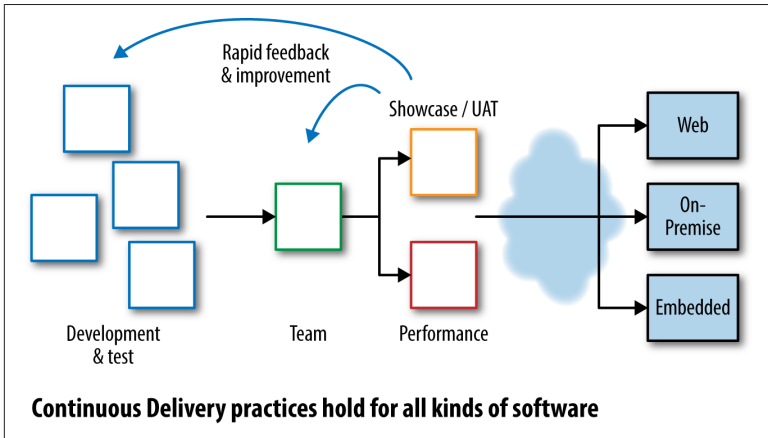


Figure 1-1. Continuous Delivery feedback

In contrast, as shown in Figure 1-2, Continuous Deployment is a push-based approach: when software developers commit a new feature to version control, it is pushed toward the Live systems automatically after successfully passing through a series of automated tests and checks. This results in many Production deployments a day.

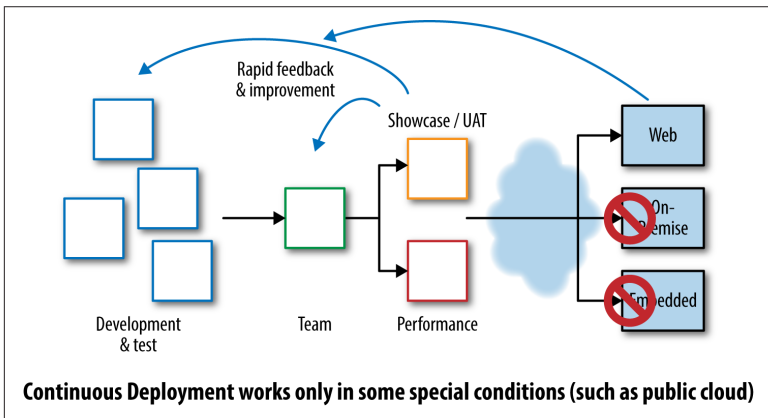


Figure 1-2. Continuous Deployment feedback

From our experience, Continuous Deployment is a niche practice useful for web-based systems in small, highly focused development

teams. Continuous Delivery suits a much wider range of software scenarios and is a much more effective approach with management: “we’re giving you the power to choose when to release new features” is quite an eye-opener for many in senior management!

The Importance of Automation for Continuous Delivery

In Continuous Delivery, we aim to automate all the repetitive, error-prone activities that humans do that can lead to inconsistent, unreliable, or unrepeatable processes and outputs:

- Software compilation and unit testing (“builds”)
- Software testing (component, service, integration, and UI)
- Deployment of software and infrastructure through all environments, including Production
- Configuration of applications and infrastructure (including networks, DNS, virtual machines [VMs], and load balancers)
- Database changes (reference data, schema changes, and data migrations)
- Approval of everyday IT changes (“standard changes”)
- The tracking and tracing of change history and authorizations
- Testing of Production systems

NOTE

We use the word “Production” to refer to the environment where your software is serving customers. This is sometimes called “Live,” but we feel this is a loaded term likely to cause confusion.

These are all areas where we can get computers to do a much better job than humans: more consistent, more repeatable, more reliable. Areas that we leave for human input are limited to those areas where humans add a huge amount of value: software development itself, test strategy and approaches, exploratory testing [Hendrickson], performance test analysis, and deployment and rollback decisions.

Why Is Continuous Delivery Needed?

Developing and operating modern software successfully requires a combination of good tools, well-trained and well-socialized people, good team discipline, a clear and shared purpose, and well-aligned financial and organizational goals. Too many organizations believe that they can do without one or more of these ingredients, but the result is that software is expensive, late, or faulty, and often all three.

By adopting Continuous Delivery, organizations can lay the foundations for highly effective software delivery and operations, consistently producing high-quality software that works well in Production and delights clients and customers.

Why Windows Needs Special Treatment

Many of the tools commonly used for Continuous Delivery are not available natively for the Windows platform, so we need to find Windows-native approaches that achieve the same ends.

In particular, package management (in the form of NuGet and Chocolatey) has only been available on the Windows platform since 2010, whereas operating systems such as Linux and BSD (and more recently Mac) have used package management for software distribution and installation since at least 1997. Other good approaches being adopted on Windows are:

- *Plain text files for configuration* of applications and services (rather than the Windows Registry or databases)
- Controlling application behavior from the *command line* (rather than a GUI)
- A *multivendor, open source-friendly approach* to software in the Windows ecosystem
- Easily scriptable *package management* with declarative dependencies

Many of the new and forthcoming features in Windows Server 2016 and Visual Studio 2015/Visual Studio Online (VSO) are designed with Continuous Delivery in mind.

NOTE

PowerShell provides a command-line interface to Windows for controlling application and system behavior and configuration. As a result, PowerShell is now the primary means of automating Windows components and .NET software.

The tips and advice in this book will help you navigate the route to Continuous Delivery on the Windows/.NET platform. Real-world case studies with Windows/.NET software show that successful companies around the world are using the techniques.

Terminology Used in This Book

We'll use these terms throughout the book:

Artifact

Immutable, versioned files or collections of files that are used for deployments

Continuous Delivery (CD)

Reliable software releases through build, test, and deployment automation

Continuous Integration (CI)

Integrating work from developers as soon as possible, many times a day

Cycle time

The time spent by the team working on an item until the item is delivered (in a Continuous Delivery context)

DBA

Database administrator

DSC

Desired State Configuration

DSL

Domain-specific language

IIS

Internet Information Services

Infrastructure

Servers, networks, storage, DNS, virtualization—things that support the platform

Infrastructure configuration management (or just config management)

See *Infrastructure as code*

Infrastructure as code

Treating infrastructure as if it were software, and using software development techniques (such as test-driven development) when developing infrastructure code

LDAP

Lightweight Directory Access Protocol

On-premise

Self-hosted systems, managed internally by the organization—the opposite of SaaS-hosted systems

Package management

Software and established practices for dependency management via named collections of files with specific version numbers

Production

The environment where the software runs and serves customers, also known as Live

SCCM

System Center Configuration Manager

Software-as-a-Service (SaaS)

Hosted software services—the opposite of on-premise or self-hosted systems

SysAdmin

System administrator

Test-driven development (TDD)

Evolving software in a safe, stepwise way, emphasizing maintainability and clarity of purpose

User interface (UI)

Also known as graphical user interface (GUI)

Version control system (VCS)

Sometimes called source code control or revision control

Version Control

Controlling versions of text files (code, configuration, static data, etc.) is a crucial facet of Continuous Delivery. Version control provides traceability, predictability, and repeatability, because we're forced to treat the version control system as the definitive source of truth. Version control is also a central communication mechanism between teams and team members, indicating the intent and purpose of our changes so that people can make better decisions.

Key Version Control Practices

Some important principles for version control in a Continuous Delivery context are:

- Commits to version control should be cohesive and meaningful, helping people to see *why* a change was made.
- Commits should happen many times per day—if your version control system or practices work against multiple daily commits, you're likely using the wrong approach.
- Branch as little as possible (see “[Branching Options](#)” on page 11)—use techniques such as feature toggles to manage partially completed work. Focus on completing a small number of changes rather than tackling many things in parallel.
- Any non-trunk branches should be short-lived—especially feature branches.

Furthermore, for Continuous Delivery, we tend to prefer many smaller repositories to one large repository, using package management to bring together dependent modules (see [“Use NuGet to Manage Internal Dependencies” on page 14](#)).

Carnect: Version Control Changes for Continuous Delivery

Carnect is a leading car rental booking engine and solutions provider working worldwide with a portfolio of over 500 car rental suppliers. Founded in Hamburg in 1999 as MicronNexus GmbH, Carnect in 2007 joined TUI Group, the largest leisure, travel, and tourism company in the world.

Carnect provides digital content via white-label websites and vehicle booking services via web APIs; their systems run on the Windows platform using a mixture of VB.NET and C# components. The Carnect teams began adopting elements of Continuous Delivery in early 2015, with a focus on build automation, automated tests, and repeatable deployments.

Andy Lole, CTO at Carnect, explains that:

The combination of .NET on Windows and tools like GitHub, JIRA, and TeamCity works really well for us. The tech teams at Carnect love the development speed and power of .NET, but using TFS for version control was really hurting our ability to release in a controlled and repeatable manner. By moving our code to GitHub and wiring it up to JIRA and TeamCity we can safely run much faster and more consistently than before.

The need to trace a change from version control through a build to a deployed package was a big factor in adopting newer practices, and this new ability has helped the teams to track down within minutes problems that used to take days to resolve.

Version Control Technologies

In 2015 and beyond, you should choose or switch to a hosted (SaaS) version control solution unless you have a very clear understanding of why you need a self-hosted (on-premise) tool. Only those organizations with an excellent capability in infrastructure and security management should consider on-premise version control. If you're certain that self-hosted/on-premise is right for you, then good

options are [GitHub Enterprise](#), [Bitbucket Server \(formerly Stash\)](#), and [RhodeCode](#).

The most effective SaaS version control providers for Windows and .NET software include:

- [GitHub](#)
- [Bitbucket](#)
- [Visual Studio Online](#)
- [CodebaseHQ](#)
- [CloudForge](#)

The SaaS hosting solution acts as the definitive central location for code integration and collaboration.

NOTE

Modern SaaS version control tools provide enterprise-level features such as integration with LDAP and fine-grained security permissions control, along with private repositories. Using a SaaS provider does not mean that your files need to be open sourced.

You will also need client tools to run on each workstation, build agent, or server. Client tools are generally free of charge.

Git

Git is currently the de facto standard for version control and is a powerful, safe, and flexible version control tool. You should choose a Git-based solution for Continuous Delivery with Windows and .NET unless you have a good reason to use a different tool, and invest in Git training for everyone using version control, including SysAdmins and DBAs.

Free client tools for Git on Windows include:

Git for Windows (or git bash)

Good for cross-platform work

GitHub for Windows

A GUI tool from GitHub

Atlassian SourceTree

Rich GUI tool for beginners and experts

POSH-git

PowerShell environment for Git

Git Extensions

Git integration for Visual Studio

TFS 2013 or later

(Using the Git option for version control)

In practice, if you choose to use Git, you will need a hosted solution such as GitHub, Bitbucket, or Visual Studio Online.

Mercurial

Mercurial is similar to Git. Some Windows and .NET-based tools support Mercurial, but with the popularity of Git and GitHub, along with Git's superior branch handling, it's unlikely that Mercurial will provide much benefit over Git for most teams.

If you need to use Mercurial on Windows, one of the best client tools is currently **Atlassian SourceTree**. SourceTree is a free download and supports Git as well as Mercurial.

Subversion

Subversion is a well-established version control system that can work well for small- to medium-sized repositories in a Continuous Delivery context. Subversion uses a central server for the definitive repository copy along with operations such as `svn log` for seeing history. This means that some operations (particularly viewing the log) are slow compared to similar operations in Git or Mercurial.

Compelling reasons to use Subversion include **TortoiseSVN**, which provides rich integration with the Windows Explorer shell, and Subversion's native support for binary files, making updates and fresh checkouts (for CI) much quicker than with Git for large numbers of binary files.

TFS

Until the release of Team Foundation Server (TFS) 2013, the Team Foundation Version Control (TFVC) component of TFS alone provided version control features using a central server. The early versions of TFS/TFVC (2005, 2008, 2010, and 2012) possessed several

features of Microsoft's legacy Visual Source Safe VCS that were incompatible with Continuous Delivery [Hammer].

However, TFS 2013 added **support for Git repositories**, thereby combining the richness of Visual Studio with the flexibility and power of Git. Future versions of TFS and Visual Studio Online will provide **support for accessing Git repositories over SSH** (currently, only Git over HTTPS is supported). With TFS 2015 there is also expanded support for the cloud-based Visual Studio Online, integration with Azure, and native integration with GitHub.

With **strong support for Git, as well as programmability via REST APIs and agile-oriented features**, TFS 2015 promises to be a much better fit for Continuous Delivery than previous versions.

Branching Options

A developer has a problem to fix, so they create a feature branch. Now they have two problems to fix.

—Anon.

Modern version control systems allow for branching of the source code, thereby allowing for multiple parallel streams of work on the same project. This is an alluring prospect that gives the illusion that multiple development streams can be achieved on the same code-base without impacting one another. Unfortunately, this method is fraught with issues when the time comes to bring these parallel streams back together.

Where possible, avoid many long-lived branches, preferring instead either trunk-based development or short-lived feature branches. **Trunk-based development** is the practice of making changes directly onto the trunk or master branch without feature branches. This practice requires a discipline in the developers to ensure that the HEAD of the trunk is always in a releasable state; a stepping-stone toward trunk-based development is **GitHub Flow**, the practice of using pull requests.



Do not be tempted to rush into branching just because your version control tool has good branching support. Git in particular supports branching well, but Git was designed for collaboration on the Linux kernel code by geographically separated individuals. Your software teams (should) have much better communication than the Linux kernel team, so give preference to communication instead of branching!

Pull Requests

Pull requests, or merge requests, are becoming a popular way to manage development. They work in a similar way to feature branches, although on a much smaller scale.

The developer works on a local clone of the codebase, making a small change that is pushed to her centrally hosted clone. Then, using the hosted tool, GitHub, Bitbucket Server (formerly Stash), or something similar, the developer makes a request to the owners of the origin repository to review and merge the change.

The process stems from open source development where developers are spread out throughout the world and in various time zones. It allows for the owners to review and judge each request as to whether it is suitable for merging. For this scenario it is perfect, although in larger organizations it can make the flow of changes quite stilted.



The **GitFlow model** for development work with Git is popular but in our view can lead to situations where CI happens less frequently than with other branching models. GitFlow also has a steep learning curve for people new to version control. We recommend that you explore other models before trying GitFlow.

Feature Toggles

Large features and changes are always needed. We can decompose the work to be as small as possible, yet there will be times when they are part of something larger that can only work together once fully implemented. This can still be achieved in trunk-based development by employing **Feature Toggles**.

Feature Toggles, at their simplest, can merely be `if` statements surrounding the usage of your new functionality. They could also be

implemented through factory methods or instantiating different implementations via **IoC**. The scope and complexity of the toggle depends on what your feature change is.



For further reading on branching strategies, look up Steve Smith's excellent blog series on **Version Control Strategies**.

Use NuGet for Dependencies

NuGet is the open source dependency package management tool for .NET. NuGet is distributed as a Visual Studio Extension and is preinstalled by default in Visual Studio version 2012 and later. NuGet can also be used from the command line, which is useful for automated scripting. The NuGet client tools provide the ability to both produce and consume packages. **NuGet Gallery** is the central package repository used by open source package authors and consumers.

NuGet is similar in its goals to package managers used in other languages, such as Java's Maven, Ruby's Gems, and Node's NPM. It provides a way to specify a package for installation and along with it installs that package's dependencies at the required versions.

As seen in **Figure 2-1**, the NuGet Visual Studio Extension provides a GUI for finding and installing packages from the NuGet Gallery.

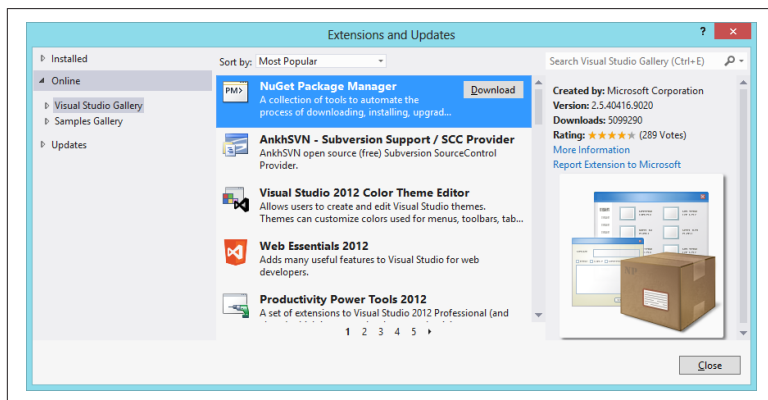


Figure 2-1. The NuGet Visual Studio Interface

The following sections cover some good practices for working with packages and package management.

Do Not Store Packages in Version Control

Store only the *packages.config* files in version control, not actual packages. Use NuGet Package Restore and the packages will be re-downloaded, from a local cache if available, to your project at compile time. This way you can prevent the binaries from bloating your version control. The restore is an MSBuild step included in the solution file and is generally supported on CI servers.

Use NuGet to Manage Internal Dependencies

Your own internally shared libraries are excellent candidates for NuGet packages. By making your shared libraries available in this way you are ensuring that the library is built only once, as is the case with all other artifacts in a Continuous Delivery pipeline.

To generate the NuGet package, simply run the following:

```
nuget pack MyProject.csproj
```

NOTE

Originally, the NuGet package format was for build-time dependencies only, although several tools now use the NuGet format for deployable artifacts too. For a clearer distinction between build-time and run-time dependencies, use **Chocolatey**, currently the de facto apt-get tool for Windows and .NET. In the future, other package managers may appear, facilitated by the emerging **OneGet packager manager management framework**.

There are various NuGet hosting applications that can be installed on premise such as Artifactory and **ProGet**, and SaaS-based tools such as **MyGet** and **Artifactory Cloud**. Some other tools, including TeamCity and Octopus, support built-in hosting of NuGet feeds; in fact, TeamCity includes an optional build step to generate NuGet packages.

Summary

Version control of all text-based files is an essential foundation for Continuous Delivery. When choosing tools for version control, it's important to consider not only developers but also people in testing and operations.

Continuous Integration

Continuous Integration is central to Continuous Delivery. Many organizations assume that they are “doing CI” if they run a CI Server (see below), but good CI is a well-defined set of approaches and good practices, not a tool. We integrate continuously in order to find code conflicts as soon as possible. That is, by doing Continuous Integration, we explicitly expect to hit small snags with merges or component interactions on a regular basis. However, because we are integrating continuously (many times every day), the size and complexity of these conflicts or incompatibilities are small, and each problem is usually easy to correct. We use a CI server to coordinate steps for a build process by triggering other tools or scripts that undertake the actual compilation and linking activities. By separating the build tasks into scripts, we are able to reuse the same build steps on the CI server as on our developer workstations, giving us greater consistency for our builds.

In this chapter, we first look at some commonly used CI servers for Windows and .NET and their strengths and weaknesses. We then cover some options for driving .NET builds both from the command line and from CI servers.

CI Servers for Windows and .NET

Although CI is a practice or mindset, we can use a CI server to help us do some heavy lifting. Several CI servers have dedicated Windows and .NET support, and each has different strengths. Whichever CI server you choose, ensure that you do not run builds

on the CI server itself, but use build agents (“remote agents” or “build slaves”; see [Figure 3-1](#)). Build agents can scale and be configured much more easily than the CI server; leave the CI server to coordinate builds done remotely on the agents.

NOTE

Remote build agents require a physical or virtual machine, which can be expensive. A possible route to smaller and cheaper build agents is [Windows Nano](#), a lightweight version of Windows announced in April 2015.

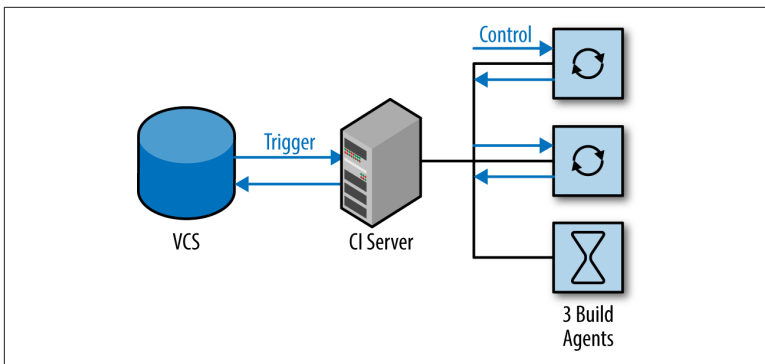
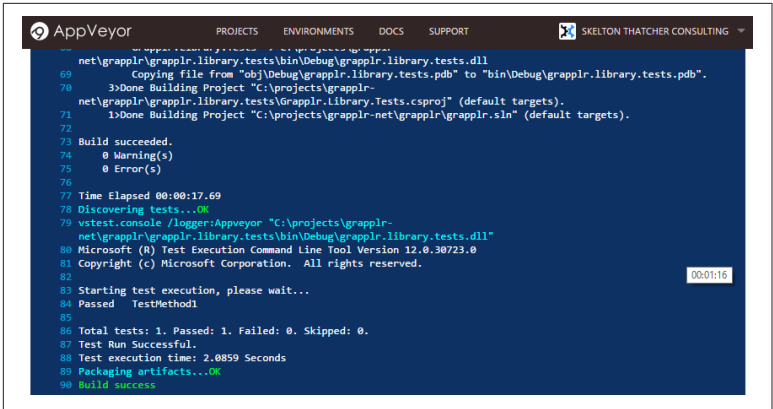


Figure 3-1. Use remote build agents for CI

AppVeyor

[AppVeyor](#) is a cloud-hosted CI service dedicated to building and testing code on the Windows/.NET platform (see [Figure 3-2](#)). The service has built-in support for version control repositories in online services such as GitHub, Bitbucket, and [Visual Studio Online](#), as well as generic version control systems such as Git, Mercurial, and Subversion. Once you have authenticated with your chosen version control service, AppVeyor automatically displays all the repositories available.

The focus on .NET applications allows AppVeyor to make some useful assumptions about what to build and run within your source code: solution files (*.sln*) and project files (*.csproj*) are auto-detected, and there is first-class support for NuGet for build dependency management.



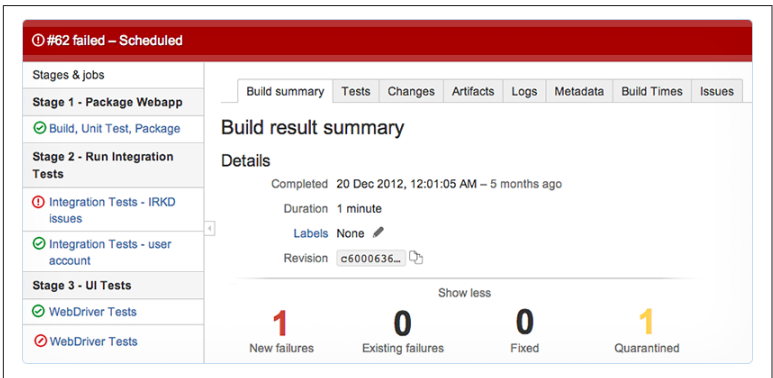
```
AppVeyor PROJECTS ENVIRONMENTS DOCS SUPPORT SKELTON THATCHER CONSULTING
69 net\grapplr\grapplr.library.tests\bin\Debug\grapplr.library.tests.dll
70 Copying file from "obj\Debug\grapplr.library.tests.pdb" to "bin\Debug\grapplr.library.tests.pdb".
71 3>>Done Building Project "C:\projects\grapplr-
72 net\grapplr\grapplr.library.tests\Grapplr.Library.Tests.csproj" (default targets).
73 1>Done Building Project "C:\projects\grapplr-net\grapplr\grapplr.sln" (default targets).
74 Build succeeded.
75 0 Warning(s)
76 0 Error(s)
77 Time Elapsed 00:00:17.69
78 Discovering tests...OK
79 vstest.console /logger:Appveyor "C:\projects\grapplr-
80 net\grapplr\grapplr.library.tests\bin\Debug\grapplr.library.tests.dll"
81 Microsoft (R) Test Execution Command Line Tool Version 12.0.30723.0
82 Copyright (c) Microsoft Corporation. All rights reserved.
83 Starting test execution, please wait... 00:01:16
84 Passed TestMethod1
85
86 Total tests: 1. Passed: 1. Failed: 0. Skipped: 0.
87 Test Run Successful.
88 Test execution time: 2.0859 Seconds
89 Packaging artifacts...OK
90 Build success.
```

Figure 3-2. A successful build in AppVeyor

If all or most of your software is based on .NET and you use cloud-based services for version control (GitHub, Bitbucket, etc.) and Azure for hosting, then AppVeyor could work well for you.

Bamboo

Atlassian Bamboo (Figure 3-3) is a cross-platform CI and release management tool with first-class support for Windows and .NET. Short build times are essential for rapid feedback after a commit to version control, and Bamboo supports remote build agents for speeding up complex builds through parallelization.



#62 failed – Scheduled

Stages & jobs

- Stage 1 - Package Webapp
 - Build, Unit Test, Package
- Stage 2 - Run Integration Tests
 - Integration Tests - IRKD Issues
 - Integration Tests - user account
- Stage 3 - UI Tests
 - WebDriver Tests
 - WebDriver Tests

Build result summary

Details

Completed 20 Dec 2012, 12:01:05 AM – 5 months ago

Duration 1 minute

Labels None

Revision c6000636...

Show less

1	0	0	1
New failures	Existing failures	Fixed	Quarantined

Figure 3-3. Bamboo builds

Along with drag-and-drop organization of build plans, Bamboo has strong integration with Atlassian JIRA to show the status of builds

against JIRA tickets, a useful feature for teams using JIRA for story tracking.

Bamboo has strong support for working with multiple branches, both short-lived and longer-term, so it may be especially suitable if you need to gain control over heavily branched version control repositories as you move toward trunk-based development.

BuildMaster

BuildMaster from Inedo can provide CI services, along with many other aspects of software deployment automation (Figure 3-4). BuildMaster is written in .NET and has first-class support for both .NET and non-.NET software running on Windows. Its key differentiators are ease of use, team visibility of build and deployment status, and incremental adoption of Continuous Delivery.

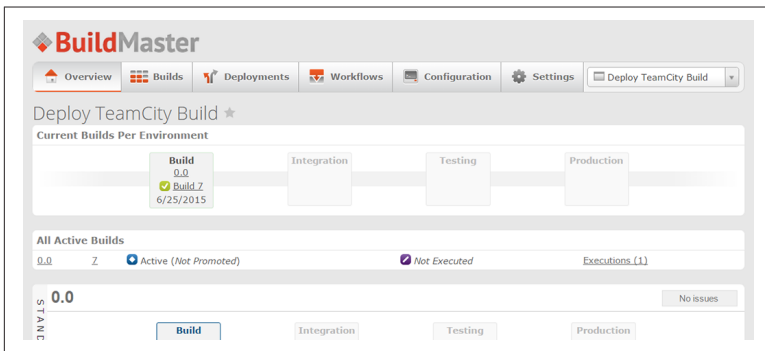


Figure 3-4. Deployments in BuildMaster

BuildMaster’s focus on incremental progress toward Continuous Delivery is particularly useful because moving an existing codebase to Continuous Delivery in one go is unlikely to succeed. In our experience, organizations should start Continuous Delivery with just a single application or service, adding more applications as teething problems are solved, rather than attempting a “big bang” approach.

GoCD

GoCD is arguably the most advanced tool for Continuous Delivery currently available. Almost alone among Continuous Delivery tools, it has first-class support for fully configurable deployment pipelines,

and we look in more depth at this capability later in the book (see [Chapter 4](#)).

GoCD can also act as a CI server, and has a developing [plug-in ecosystem](#) with support for .NET technologies like NuGet. The UI is particularly clean and well designed. Nonlinear build schemes are easy to set up and maintain in GoCD, as seen in [Figure 3-5](#).

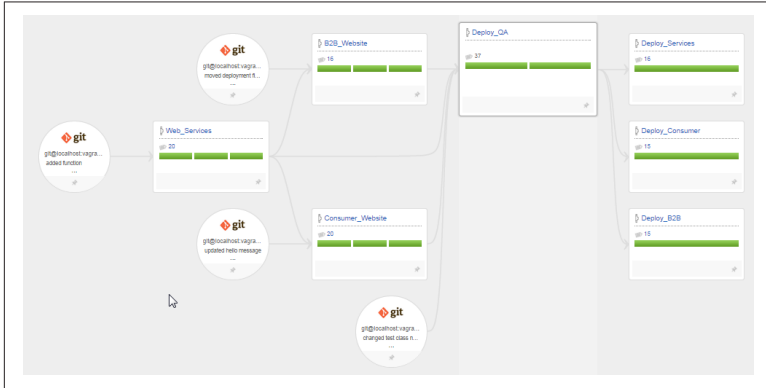


Figure 3-5. Nonlinear build flows in GoCD

GoCD has strong support for parallelization of builds using build agents. This can lead to a huge reduction in build times if a build has multiple independent components (say, several *.sln* solution files, each generating separate class libraries or services).

Parallel builds, a clean UI, and support for nonlinear build flows make GoCD an effective choice for CI for more complex situations.

Jenkins

[Jenkins](#) is a well-established CI server, likely the most popular one for Linux/BSD systems. One of the most compelling features of Jenkins is the plug-in system and the many hundreds of plug-ins that are available for build and deployment automation.

A very useful feature of Jenkins is the “weather report” ([Figure 3-6](#)), which indicates the trend of builds for a Jenkins job: sunshine indicates that all is well, whereas rain indicates that the build job is regularly failing.



Figure 3-6. Jenkins build jobs indicate long-term health

A downside of using Jenkins as a CI for .NET builds is that the support for .NET ecosystem technologies such as NuGet or Azure is largely limited to the use of hand-crafted command-line scripts or early-stage plug-ins. Your mileage may vary!

TeamCity

JetBrains provides the commercial product TeamCity, likely the most popular in use with .NET projects on Windows. Compared to other CI tools, the highly polished UI is one of its best features.

TeamCity uses a hierarchical organizational structure—projects contain subprojects, which in turn contain multiple builds. Each build consists of an optional version control source, trigger rules, dependencies, artifact rules, and multiple build steps.

A build can run a single build step, which triggers a script, or multiple build steps can be specified to run in sequence to compose the required steps. TeamCity includes preconfigured build steps for many common actions, such as compiling a Visual Studio solution and generating NuGet packages.

A running build will display a live log of progress and a full history kept for a configurable amount of time.

NOTE TeamCity includes support for triggering builds from branches and pull requests merged into the HEAD of trunk. Using this feature allows for testing of a pull request before finally merging.

TeamCity also exposes a REST API allowing for scripting and a library of over 100 plug-ins available to use.

TFS Build / VSO

Team Foundation Server Build (often known as TFS) is Microsoft's CI and deployment solution and has undergone several iterations, the most recent being TFS 2015 (self-hosted) or Visual Studio Online (VSO). The versions of TFS prior to TFS 2015 are “notoriously unfriendly to use” [Hammer] and have very limited support for non-developers using version control and CI because in practice they require the use of Visual Studio, a tool rarely installed on the workstations of DBAs or SysAdmins. The pre-2015 versions of TFS Build also used an awkward XML-based build definition scheme, which was difficult for many to work with.

However, with **TFS Build 2015** and **Visual Studio Online**, Microsoft has made a significant improvement in many areas, with many parts rewritten from the ground up. Here is the advice from the Microsoft Visual Studio team on using TFS 2015/VSO for CI:

If you are new to Team Foundation Server (TFS) and Visual Studio Online, you should use this new system [TFS 2015]. Most customers with experience using TFS and XAML builds will also get better outcomes by using the new system.

The new builds are web- and script-based, and highly customizable. They leave behind many of the problems and limitations of the XAML builds.¹

—Visual Studio team

TFS Build 2015 supports a wide range of build targets and technologies, including many normally associated with the Linux/Mac platform, reflecting the heterogeneous nature of many technology stacks these days. The CI features have been revamped, with the live build status view (shown in **Figure 3-7**) being particularly good.

¹ <https://msdn.microsoft.com/en-us/Library/vs/alm/Build/overview>

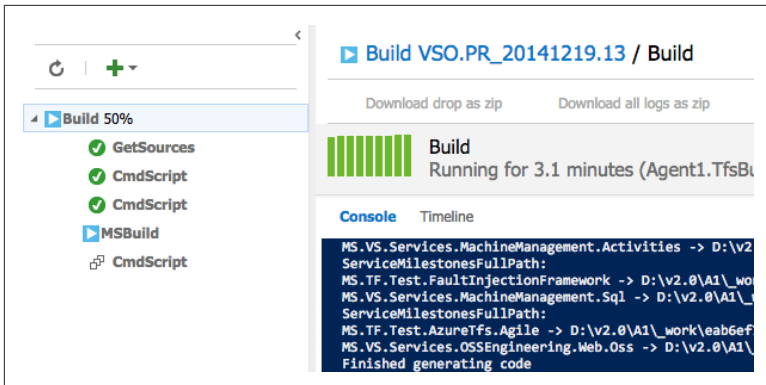


Figure 3-7. Live build status view in TFS 2015

Crucially for Continuous Delivery, TFS Build 2015 uses standard build scripts to run CI builds, meaning that developers can run the same builds as the CI server, **reducing the chances of running into the problem of “it builds fine on my machine but not on the CI server”**. TFS Build 2015/VSO appears to have very capable CI features.

Build Automation

Build Automation in .NET usually includes four distinct steps:

Build

Compiling the source code

Test

Executing unit, integration, and acceptance tests

Package

Collating the compiled code and artifacts

Deploy

Getting the package onto the server and installed

If any of the steps fails, subsequent steps should not be executed; instead, error logs should be available to the team so that investigations can begin.

Build Automation Tools

The various CI servers described in the previous chapter coordinate steps for a build process by triggering other tools or scripts. For the step of building and compiling source files there are various options in the .NET space. A description of a select few follow.

MSBuild

MSBuild is the default build toolset for .NET, and uses an XML schema describing the files and steps needed for the compilation. It is the process with which Visual Studio itself compiles the solutions, and MSBuild is excellent for this task. Visual Studio is dependent on MSBuild but MSBuild is not dependent on Visual Studio, so the build tools can be installed as a distinct package on build agents, thereby avoiding extra licensing costs for Visual Studio.

The solution (*.sln*) and project (*.csproj*) files make up almost the entirety of the scripts for MSBuild. These are generally auto-generated and administered through Visual Studio. MSBuild also provides hooks for packaging, testing, and deployment.



TIP

Builds with MSBuild can be speeded up significantly using the `/maxcpucount` (or `/m`) setting. If you have four available CPU cores, then call MSBuild like this: `msbuild.exe MySolution.sln /maxcpucount:4`, and MSBuild will automatically use up to four separate build processes to build the solution.

PSake

PSake is a PowerShell-based .NET build tool that takes an imperative “do this” approach that is contrary to the declarative “this is what I want” approach of NAnt. Several emerging .NET Continuous Delivery tools use PSake as their build tool.

Ruby/Rake

Rake saw great popularity as a build automation tool of choice from late 2009. At the time, it was a great improvement over MSBuild and NAnt—the previous popular tools. With NAnt being XML-based, it’s extremely difficult to write as a script (e.g., with control statements such as loops and ifs). Rake provided all of this, as it is a task-based DSL written in Ruby.

The downside of Rake is that Ruby is not at home in Windows. Ruby must be installed on the machine and so a version must be chosen. There's no easy way to manage multiple Ruby versions in Windows so the scripts tend to be frozen in time and locked to the version that has been installed.

NAnt

NAnt is a tool that used to be popular. It is XML-driven and unwieldy. Replace it with another tool if NAnt is still in use.

Batch Script

If the CI server you are using does not support PowerShell and other tools such as Ruby cannot be installed, then batch scripting can be used for very small tasks. It is not recommended and can quickly result in unmanageable and unreadable files.

7digital: Continuous Integration for Continuous Delivery

7digital is the power behind innovative digital listening experiences. Music streaming and download services are powered by their API technology—an offering that includes everything needed to create and run streaming and download products, including easy payment transactions, user management, and locker storage.

7digital's RESTful API platform is largely powered by Windows and .NET technologies. This platform provides access to catalog metadata, streaming, and downloading of music files. The API is composed of smaller, internal APIs, each focused on a specific area and developed by product-focused teams.

The API development teams use a trunk-based approach to source control. Commits are made to the trunk and feature branches are heavily discouraged. Their deployment pipeline is fast and trustworthy so that releases happen regularly and as such keep the amount of change in each release as small as possible. Rollbacks are also an essential factor in the deployment pipeline, which allows 7digital to quickly recover from any issues with a release.

Paul Shannon, VP Technology at 7digital, says:

We use a combination of TeamCity and Octopus to facilitate a simple and effective automated deployment process. This permits us to move quickly, confidently deploying the changes we

push into the trunk. Our approach to always be confident that our code integrates cleanly has significantly reduced the amount of time code is in the development pipeline before it is released, which provides value to our customers more quickly and reduces risk.

This approach was adopted in 2008 as part of a wider strategy toward Continuous Delivery. The larger part of this strategy involved breaking down the original monolithic API into the smaller components, allowing the teams to move faster. It also included a focus on improving strategies for testing, which in turn gave the developers faith in their code changes.

Integrating CI with Version Control and Ticket Tracking

Most CI servers provide hooks that allow for integration with version control and ticketing systems. With these features we can enable the CI server to trigger a build on the commit of a changeset, to link a particular changeset with a ticket, and to list artifacts generated from builds from within the ticket.

This allows the ticket tracker to store a direct link to code changes, which can be used to answer questions relating to actual changes made when working on this ticket.

Patterns for CI Across Multiple Teams

Effective CI should naturally drive us to divide the coding required for *User Stories* into developer- or team-sized chunks so that there is less need to branch code. Contract tests between components (with **consumer-driven contracts**) help to detect integration problems early, especially where work from more than one team needs to be integrated.

Clear ownership of code is crucial for effective Continuous Delivery. Generally speaking, avoid the model of “any person can change anything,” as this model works only for highly disciplined engineering-driven organizations. Instead, let each team be responsible for a set of components or services, acting as gatekeepers of quality (see “**Organizational Changes**” on page 55).

Architecture Changes for Better CI

We can make CI faster and more repeatable for .NET code by adjusting the structure of our code in several ways:

- Use smaller, decoupled components
- Use exactly one *.sln* file per component or service
- Ensure that a solution produces many assemblies/DLLs but only one component or service
- Ensure that each *.csproj* exists in only one *.sln* file, not shared between many
- Use NuGet to package internal libraries as described in “[Use NuGet to Manage Internal Dependencies](#)” on page 14

NOTE

The new *project.json* project file format for DNX (the upcoming cross-platform development and execution environment for .NET) looks like a more CI-friendly format compared to the traditional *.csproj* format. In particular, the *Dependencies* feature of *project.json* helps to define exactly which dependencies to use.

Cleanly separated libraries and components that express their dependencies via NuGet packaging tend to produce builds that are easier to debug due to better-defined dependencies compared to code where projects sit in many solutions.

Summary

Continuous Integration is the first vital step to achieving Continuous Delivery. We have covered the various options for CI servers, version control systems, build automation, and package management in the Windows and .NET World.

When choosing any tools, it is important to find those that facilitate the desired working practices you and your team are after, rather than simply selecting a tool based on its advertised feature set. For effective Continuous Delivery we need to choose Continuous Integration tooling that supports the continuous flow of changes from development to automated build and test of deployment packages.

Deployment Pipelines

Deployment pipelines are a key part of Continuous Delivery. A deployment pipeline is a series of steps that occur between CI and Production deployment, coordinated by a software tool. We use a deployment pipeline for several reasons:

- To automate the different build, test, and deployment activities
- To visualize the progress of software toward Production
- To find and reduce bottlenecks or wait times in the deployment process

Done well, a deployment pipeline acts as a realization of part of a **Value Stream Map**, which makes the deployment pipeline a useful tool for sharing information with people in the organization who are familiar with Value Stream Mapping as a key part of modern business practices. Many people simply reuse their CI tool for managing their deployment pipeline, but this may omit the valuable orchestration and visualization that more specialized deployment pipeline tools provide.

Mapping Out a Deployment Pipeline

To start with, build a so-called “walking skeleton” deployment pipeline, with each stage between code commit and Production represented as a simple echo `hello, world!` activity.

As described by Alistair Cockburn on alistair.cockburn.us:

A Walking Skeleton is a tiny implementation of the system that performs a small end-to-end function. It need not use the final architecture, but it should link together the main architectural components. The architecture and the functionality can then evolve in parallel.

As you find time to automate more and more of the pipeline, you'll reduce manual steps while still retaining the coordination of the tool.

Tools for Deployment Pipelines

The following sections provide an overview of a few tools that work well with Windows and .NET to orchestrate and visualize deployment pipelines.

GoCD

GoCD from ThoughtWorks is by far the most advanced tool for Continuous Delivery deployment pipelines. GoCD was designed specifically to support Continuous Delivery deployment pipelines and offers sophisticated features such as the Value Stream Map (seen in [Figure 4-1](#)), **diamond dependencies**, massively parallel builds (using remote build agents), NuGet packages as a build trigger, and an intuitive user interface that is friendly enough for developers, IT operations people, and commercial/product people alike.

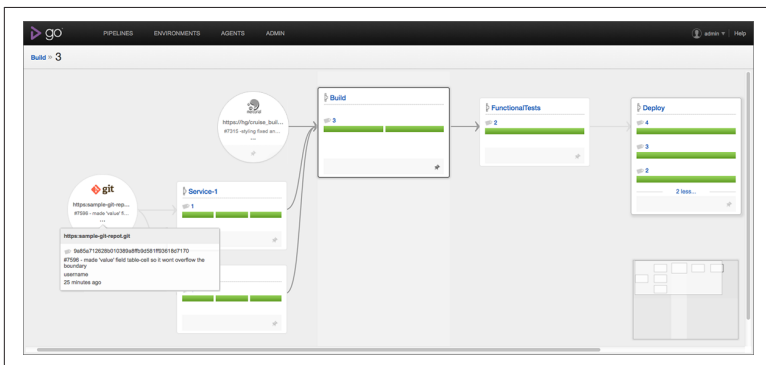


Figure 4-1. GoCD Value Stream Map view

A powerful feature for larger organizations is the rich role-based security model, which allows us to model permissions that reflect team security boundaries: for regulated sectors where developers

cannot deploy to Production, for example, this is a major enabler. GoCD also provides traceability for the activity in each pipeline stage, showing what or who triggered a step. Rollbacks with GoCD are as simple as a single button click.

GoCD inherently supports *nonlinear deployment pipelines*, which enables teams to make informed decisions about what tests to run before deployment; this contrasts with some tools that encourage software to pass through many stages or environments every time. For large, complicated .NET software systems, GoCD is a major enabler for Continuous Delivery.

Octopus

Octopus is a deployment tool for .NET with a flexible plug-in system based on PowerShell. It uses NuGet packages as its artifact format and can connect to remote NuGet feeds to find and deploy NuGet packages. Octopus has special features for handling .NET applications, IIS, Windows Services, and Azure, so many deployment tasks are very straightforward. As seen in [Figure 4-2](#), the concept of environments in Octopus means that deployment scripts can easily be targeted to all environments or just, for example, Production.

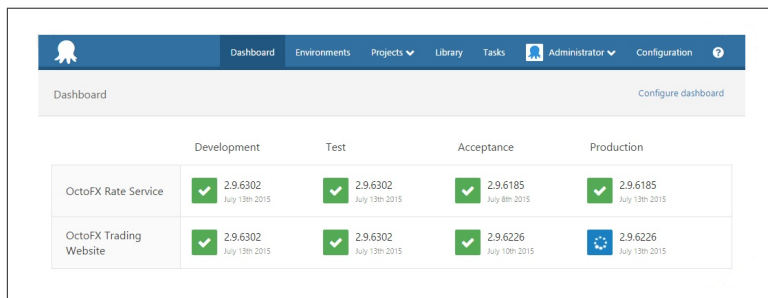


Figure 4-2. Octopus showing several environments

With its REST API, Octopus integrates well with other API-driven Continuous Delivery tools such as TeamCity and GoCD; in these cases, TeamCity or GoCD typically handles the CI and deployment pipeline coordination, calling into Octopus to carry out the Windows-specific or .NET-specific deployment tasks.

TeamCity

TeamCity is a CI server, but has many features that make it useful for deployment pipelines, including native NuGet support, powerful artifact filtering, and integration with a large number of other tools. Also, as seen in [Figure 4-3](#), many tools integrate well with TeamCity, such as Octopus, making a cohesive ecosystem of tools for Continuous Delivery.

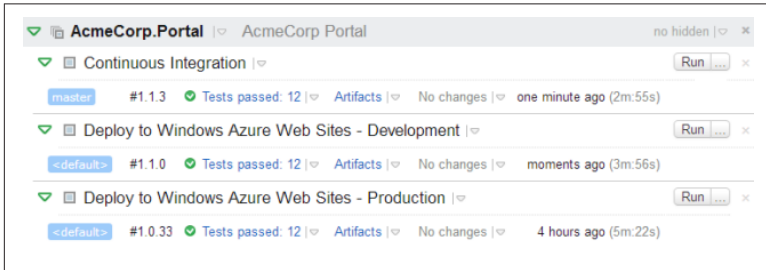


Figure 4-3. TeamCity Build Chains

TeamCity uses build chains to provide deployment pipelines, with downstream stages triggered by a successful upstream stage.

VSO

Microsoft's **Visual Studio Online** suite offers a release management workflow based on deployment pipelines, as seen in [Figure 4-4](#).¹

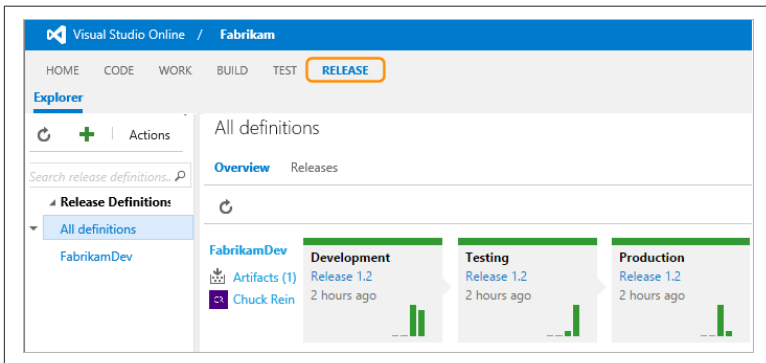


Figure 4-4. Visual Studio Online release management workflow

¹ The deployment pipeline feature is due to be available in an update to TFS 2015.

VSO offers clean, browser-based tools that provide useful metrics, custom deployment workflows, support for on-premise deployments, and deep integration with Azure.

Other Tools

Inedo BuildMaster has support for .NET-based deployment pipelines using *chained builds*, as well as built-in features for dealing with change control rules.

Jenkins has a **Delivery Pipeline Plugin** that provides a visual map of a deployment pipeline.

IBM UrbanCode Deploy is a tool for automating application deployments designed to facilitate rapid feedback and Continuous Delivery in agile development.

XebiaLabs XL Deploy offers point-and-click configuration of deployment pipelines combined with advanced reporting for regulatory compliance.

LateRooms: Deployment Pipelines for Continuous Delivery

LateRooms.com is the United Kingdom's leading hotel booking specialist, providing discounted accommodation throughout the UK, Europe, and the rest of the world, with more than 150,000 listed properties worldwide. The LateRooms technology stack is largely based on Windows and .NET (with Node.js on Linux for some frontend parts). By early 2012, the existing monolithic .NET application and release process had become cumbersome. Merging the code from multiple branches took up to six weeks, and the merged code would take a full week to deploy. Waiting months for a new feature was hurting the business; changes were needed because rival websites were adding features more frequently, and the hotel booking market is highly competitive.

In early 2012, the tech teams decided to make a major series of changes in two parts over the next three years. First, they split the codebase into smaller, discrete chunks, improving the build and deployment automation, which reduced the cycle time from months down to days or weeks (depending on the component). They then focused on automating their infrastructure provisioning and deployment (initially a combination of physical machines and

VMware ESXi/vSphere machines). This second stage further reduced cycle time to the point where code committed to version control could be live in production within a few hours—a massive 700x speed improvement over a few years before.

Steve Elliott, Development Lead at LateRooms.com, notes that:

Automating our builds and deployments has massively reduced rollbacks by allowing us to roll forwards much easier, and since we've introduced more infrastructure automation, inconsistencies between environments have been minimised as well. Using MSDeploy has worked well for us for .NET deployments, some odd cases aside. Being able to consistently configure IIS without having to do it manually has helped a lot of the teams using it.

The teams use GoCD for builds and deployment pipelines, and are currently experimenting with using a separate Continuous Integration server for the CI part of each team's builds. Source code was stored in Subversion, but a migration to GitHub is under way to allow the teams to take advantage of the rich workflow provided by Git and GitHub.

Deployment Techniques

There is more to deployments in Continuous Delivery than simply copying a package to a server. Our aim is seamless deployments without downtime and to achieve this we need to make use of various techniques, which we cover in this section.

Use Blue-Green Deployment for Seamless Deploys

With blue-green deployment, a Production environment is split into two equal and parallel groups of servers. Each “slice” is referred to as blue or green and initially only one half receives live traffic (Figure 4-5). The flow of traffic can be controlled by a programmed routing service, a load balancer, or a low-TTL DNS weighting between nodes.

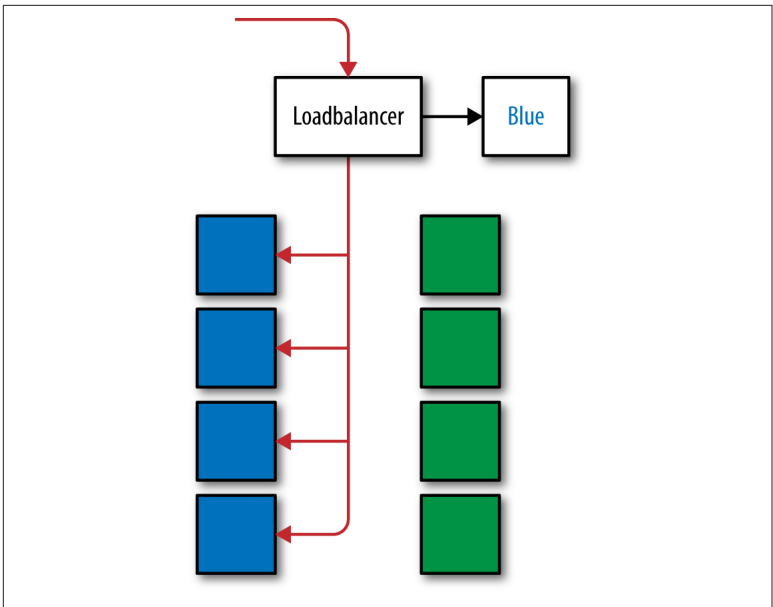


Figure 4-5. Traffic being routed to the blue slice via configuration

The slice not receiving live traffic can be used for preproduction testing of the next version of the code. Because both halves of Production are identical, this is a safe way to test a code release. The process of going live is to switch the flow of traffic from one slice to the other.

With blue-green, application rollbacks are simple and seamless; merely switch the flow of traffic back to the previously live slice.

Canary Deployments

A **Canary**, or Pilot, deployment is when prerelease/beta changes are deployed to a very small subset of the Production instances, possibly only a single instance, to serve live traffic. This allows for a sanity check of the prerelease before promoting it to the rest of the Production servers. Canary releases catch issues with a release package such as misconfigurations and unrealized breaking changes. They also allow for a final verification of how a new feature will behave under live traffic while maintaining minimal impact on the rest of the platform. These instances are heavily monitored with specific thresholds—such as not having an increased error rate—defined in advance that must be achieved to allow promotion of the prerelease.

With load balancing of requests, you can tweak how much traffic the canary server receives—a low weighting would send a fraction of the traffic handled by the other servers. There should only be one Canary version being tested at a time so as to avoid managing multiple versions in Production, and it should be quick and easy to remove or roll back the instance in the event of a bad release.

A Dark Deployment is similar, except that it is not included in the load balancer and does not receive live traffic. Instead, it allows for smoke tests to be run using Production configuration before promoting the prerelease. This has less risk of affecting consumers, but does not verify performance with live traffic.

Postdeployment Checks

It is important that the deployment pipeline includes methods to rapidly verify that a deployment has been successful. Deployment verification tests (DVTs) should be kicked off immediately after a deployment package has been released to the server. These checks are very lightweight and are centered around ensuring that the mechanics of a deployment were successful, while smoke tests, as described later, are to ensure the application itself is behaving as expected.

Tests should include verifying the files are in the expected location, application pools exist with the correct settings, services have been started, and that a health check URL can be successfully hit. If these checks fail, then the deployment was not a success and should be investigated further. These checks should highlight if the issue was caused by a bug in the deployment process itself, in which case a rollback may not fix the issue. The tests should help locate the failure in the process. As such, DVTs should also be run when deploying to preproduction environments as tests for your deployment process.

Smoke Tests

Smoke tests are a very small subset of your automated end-to-end tests that are run against the environment after a deployment, for example, signing up a user and performing a checkout, or processing a single batch request. These verify that the main path of the application is successful. They differ from the DVTs in that they exercise the system as an end user would.

It is important to resist the temptation to run your full suite of end-to-end tests as smoke tests. The value of these tests is in a fast confirmation that the system is operating as expected after deployment rather than fully testing the application itself.

Decouple File Delivery from Code Availability

In Continuous Delivery, it is useful to split the deployment of new software into two separate parts:

- Delivery of files onto a server
- Making the new software active

On Windows, we have several options for this. If our software runs in IIS, we can deliver the new code to a new folder on disk, and then simply repoint the IIS Virtual Directory to the new folder in order to make the new code live. For non-IIS code, we can use **NTFS volume mount points** to control the location where software is presented to the operating system, taking advantage of the **Add-PartitionAccessPath** and **Remove-PartitionAccessPath** PowerShell cmdlets to automate the reconfiguration of the mount points.

Script a Rollback Procedure Too

An excellent way to ensure confidence in your Continuous Delivery process is to have a simple and scripted process for returning to a known good state—a quick rollback procedure.

The most effective rollback is simply to redeploy the previous software version using the deployment pipeline. By redeploying the previous artifacts you are releasing proven code and not that which may contain an untested hotfix hastily added under pressure. After a rollback has been deployed, the DVTs and smoke tests should be run to ensure the rollback was successful. In order for rollbacks to work, each release needs to be incremental, avoiding breaking changes and supporting backward compatibility.

Automated Testing of Database Changes

The speed and volume of changes to modern software systems means that making database changes manually is risky and error-prone. We need to store database scripts, reference data, and configuration files in version control and use techniques such as TDD, CI,

and automated acceptance testing to gain confidence that our database changes will both work correctly *and* not cause the loss of valuable data when deployed to Production.

NOTE

Consider that different approaches to database change automation may be useful at different stages of a system's lifecycle: large-scale refactorings are better-suited to database comparison tools, whereas steady, ongoing development work may work well with a migrations approach. Each approach has merits in different circumstances.

Database Unit Testing

Many teams working with SQL Server use stored procedures to implement business logic in the data tier. It can be difficult to practice Continuous Delivery with significant business logic in the database; however, we can make stored procedures easier to test by using tools such as **tSQLt**, **MSTest**, **SQL Test**, and **DBTestDriven**. By running unit tests against stored procedures, these tools help us to practice TDD for database code, reducing errors and helping us to refactor more easily.

We can also make use of **LocalDB**, a lightweight, developer-friendly version of SQL Server designed for testing, which can remove the need for a full SQL Server installation.

EF Code-First Migrations

Microsoft's Entity Framework (EF) provides C#-driven migrations (called **EF Code-first Migrations**). Migrations are written in C# and can be exported as SQL for review by a DBA. If you're using EF6, then EF Code-first Migrations could work well.

FluentMigrator

FluentMigrator is an open source .NET database migrations framework with excellent support for many different databases, including SQL Server, Postgres, MySql, Oracle, Jet, and Sqlite. Installation is via a NuGet package, and migrations are written in C#. Each migration is explicitly annotated with an integer indicating the order in which the migrations are to be run.

If database changes are driven largely by developers, or if you need to target several different database technologies (say SQL Server, Postgres, and Oracle), then FluentMigrator is a good choice.

Flyway

Flyway is an advanced open source database migration tool. It strongly favors simplicity and convention over configuration. Flyway has APIs for both Java and Android and several command-line clients (for Windows, Mac OSX, and Linux).

Flyway uses plain SQL files for migrations, and also has an API for hooking in Java-based and custom migrations.

Notably, Flyway has support for a wide range of databases, including SQL Server, SQL Azure, MySQL, MariaDB, PostgreSQL, Oracle, DB2, Redshift, H2, Hsql, Derby, SQLite, and others.

Redgate Tools

Redgate provides a variety of tools for SQL Server aimed at database change and deployment automation, including SQL Source Control, ReadyRoll (for SQL migrations), SQL CI (for automated testing), SQL Release (for release control via Octopus), and DLM Dashboard (for tracking database changes through environments).

The Redgate tool suite arguably provides the greatest flexibility for database Continuous Delivery. DBA-driven changes via SSMS are supported with SQL Source Control, while developer-driven changes can use a declarative SSDT approach or ReadyRoll migrations. For organizations working with .NET and SQL Server that need flexibility and choice about how database changes are made and deployed, especially those that need input from DBAs, Redgate tools work extremely well.

SSDT

SQL Server Data Tools (SSDT) is Microsoft's declarative approach to database change automation. SSDT is aimed at developers using Visual Studio and has tight integration with other Microsoft tooling, particularly tools for Business Intelligence (BI). Some teams successfully use SSDT in combination with other tools (Redgate, tSQLt, etc.) to compensate for some of the features SSDT lacks (such as reference data management).

Other

DBMaestro provides tools for SQL Server and Oracle, with a strong focus on policy enforcement. If you're working in a regulated industry, the DBMaestro tools could work well.

Summary

We have looked at a collection of tools that work well for visualizing and orchestrating a Continuous Delivery deployment pipeline. A well-implemented deployment pipeline really helps to elevate the understanding of the progress of changes toward Production across the organization. We have also outlined some techniques for ensuring that our deployments are seamless and reliable.

Monitoring, Metrics, and APM

The CI and deployment pipeline tests give us confidence that our change will work in Production. This confidence comes from a combination of functional testing, covering the user-facing behavior of the system, and technical/performance testing, covering operational aspects of the system. However, after we have released our code to Production, how do we verify that the state of your system is still good and has not degraded? This is where monitoring comes in.

Performance Counters Are Insufficient

Windows provides **Performance Counters** to allow monitoring of server health. These standard counters will provide information such as memory and CPU usage and some generic .NET framework information such as number of exceptions thrown and time spent in garbage collection. However, these do not tell you if your app is working as expected, only whether the machine is under any load, and custom performance counters are tricky to work with.

Record Application Metrics

Open source tooling can be used very effectively for gathering application metrics. StatsD and Graphite are a common pairing; hosted on Linux, they are cost-effective to set up.

StatsD listens for statistics, like counters and timers, sent over UDP or TCP, and sends aggregates to the Graphite backend. Graphite (**Figure 5-1**) provides querying and graphing tools to visualize the

metrics. Many open source .NET clients exist that make it simpler to add StatsD metrics to your applications; they can be found in the NuGet package collection at <http://nuget.org/>.

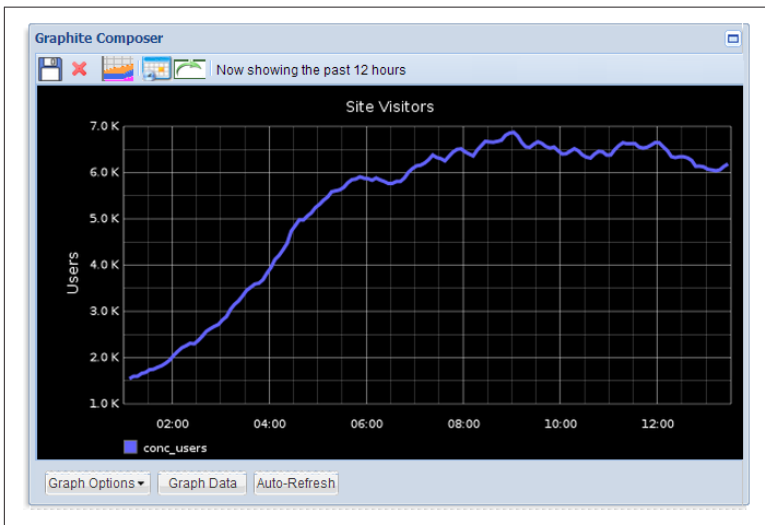


Figure 5-1. Graphite Composer



For further reading on Graphite, refer to the book *Monitoring with Graphite* by Jason Dixon.

Cloud-hosted metrics and monitoring tools such as **DataDog**, **Data-Loop**, and **ServerDensity** provide native support for Windows servers and can easily be integrated with other tools. In particular, **UpGuard** (formerly **ScriptRock**), as shown in **Figure 5-2**, has powerful capabilities for detecting and visualizing configuration differences.

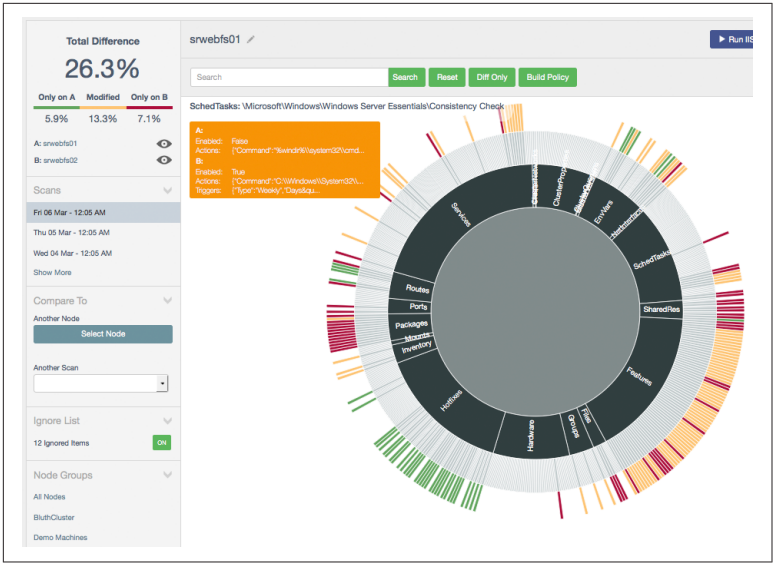


Figure 5-2. UpGuard (formerly ScriptRock)

Application metrics collected and analyzed with modern tools can be a very powerful way of understanding the inner workings of your platform.

APM Tools Can Complement Monitoring

Application Performance Management (APM) tools such as New-Relic and AppDynamics use agents installed on your servers to collect monitoring data, which is sent back to a central server. They are great as a way to get immediate insight into a Production platform; they automatically collect and aggregate data on server health and application health. They hook into the .NET runtime and collect more detail about the application, such that if an exception is thrown, it can collect a stacktrace. They allow for digging into performance issues, including highlighting potentially slow database queries. They can also provide alerts based on the data collected.

In a distributed system they will automatically add correlation data allowing tracing of a single request through all components that have had the APM agent installed. This allows the tool to build up a picture of the platform and highlight any particularly non-performant components.

Use Developer Metrics Tooling

For web applications, you can get powerful insights into your application’s performance and behavior by using **Glimpse** (Figure 5-3), which traces a wide range of execution details and presents them in a “head-up display.”

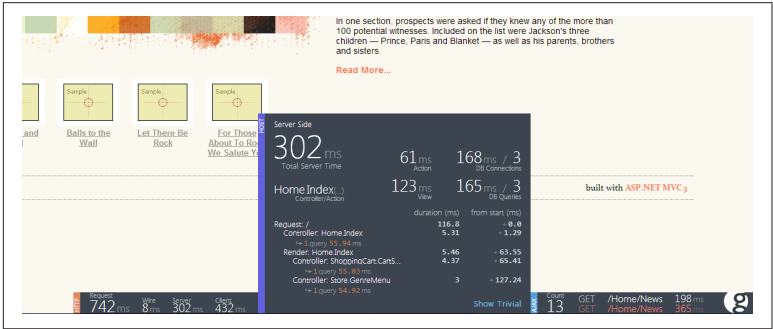


Figure 5-3. Glimpse for development metrics

The metrics include detailed statistics on HTTP, server-side metrics (including SQL execution), and AJAX (asynchronous JavaScript).

With Visual Studio 2015 and later you can use the **Application Insights** service from Microsoft to instrument and monitor your web application (Figure 5-4).

Application Insights can also monitor non-.NET applications including **J2EE Java, JavaScript, Python, and Ruby**.

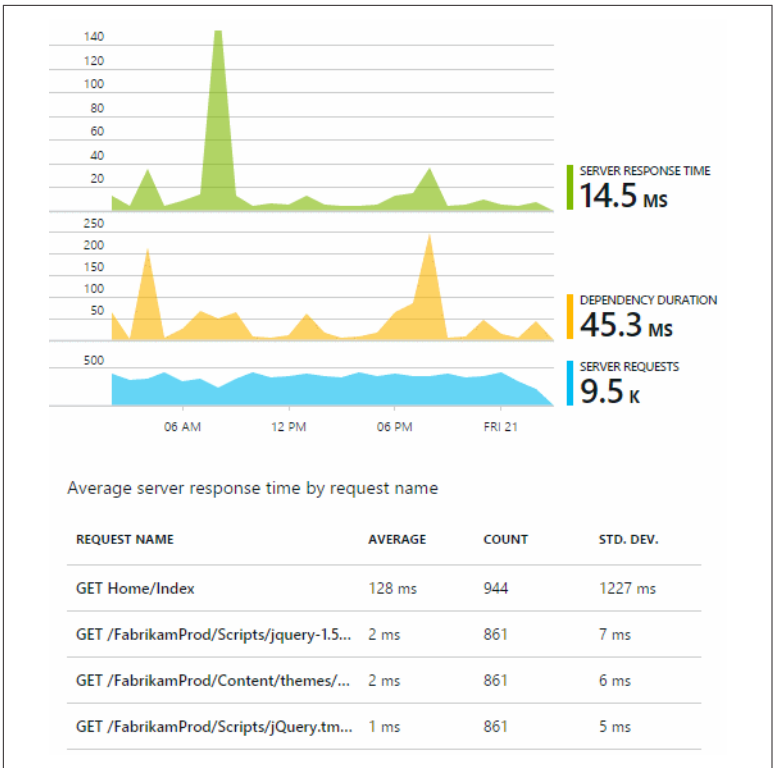


Figure 5-4. Application Insights

Aggregate Application and Windows Event Logs from All Machines

Historically, application logs were often used to record only errors and exceptional conditions. However, with the power and features of modern log aggregation tools, we can use logging as a way to gain deep insights into how our software is operating (Figure 5-5). In a Windows context, we can aggregate (bring together) logs from the Windows Event Log subsystem and our own application logs written with frameworks such as **log4net** or **Serilog** and search all the log entries using a web browser.

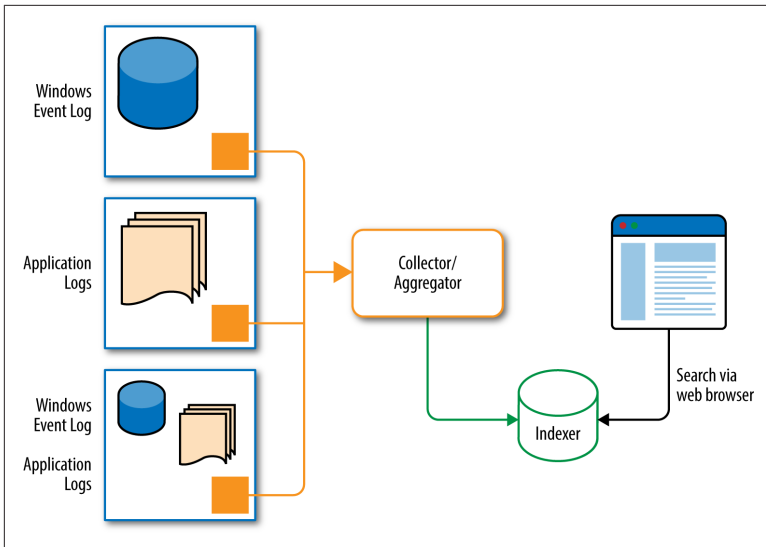


Figure 5-5. Log aggregation

The ability to search across all logs for time-coincident events or a specific text string is hugely powerful, especially if we use log aggregation tools that are available in upstream environments. In fact, if we use log aggregation tools on developer machines, we can reduce our reliance on the debugger for some diagnostic activities.

There are several log aggregation tools that work well with Windows and .NET, including:

- **ELK** (Elastic Search, LogStash, Kibana; on premise): Use **NxLog** as a log forwarder (or the default forwarder if Java is available)
- **LogEntries** (hosted): Provides good integrations with other tools alongside metrics capture
- **Loggly** (hosted)
- **NewRelic** (hosted)
- **Papertrail** (hosted)
- **Seq** (on premise or hosted): Good for structured event data

It's also worth investigating **Serilog** (possibly together with **Seq**), an advanced .NET logging framework that uses **structured logging** to

capture more detailed and meaningful log output. Serilog replaces log4net as the logging library for application code.

JUST EAT: Utilizing Metrics for Successful Continuous Delivery

JUST EAT is the world's leading online takeaway ordering service. With almost 1,200 orders placed per minute at peak times in the UK and just 45 minutes on average from checkout to delivery, the platform has to be designed from the ground up to be resilient and scalable.

JUST EAT's platform is primarily .NET/C#, hosted on Windows. It is entirely hosted in Amazon Web Services (AWS) and uses a combination of SQL Server and DynamoDB as datastores. The platform is shipped by a number of loosely coupled, small, cross-functional teams. The teams are responsible for their aspects of the product, from editor through to operating in Production. This includes the developers supporting their code out of hours.

In an increasingly distributed system, this was necessary. Recognizing this, JUST EAT has invested a significant amount of time and effort into achieving their Continuous Delivery pipeline, supported by near real-time monitoring (Statsd, Graphite, Grafana), alerting (Seyren), and centralized logging (ELK stack) to make it reasonable for engineers to debug in Production where necessary.

This culture of operations has seen JUST EAT through a very successful IPO in early 2014 and high growth since then. The JUST EAT platform is more reliable and improving constantly, at the same time as confidently shipping ever more changes to help feed hungry customers. This gives JUST EAT a competitive advantage.

Peter Mounce, Senior Software Engineer at JUST EAT, explains:

Full stack developers are recruited, then molded into engineers as they learn the skills required to not only ship software into production, but make sure it stays there peacefully. Structured logs are preferred over debuggers. Metrics are reviewed regularly; dashboards are designed for context, but not as the end result. Rather, automated alerts are designed from the point of view of their being continuously running, environment-independent tests. Automation is written so that as far as possible, the computer does the work. Failure is *expected*, and recovery modes are designed in. Incidents are seized upon as learning opportunities. Continuous, data-driven experimentation is starting to take hold.

Summary

Using dedicated tools for metrics collection, APM, and log aggregation, we can continuously verify that the state of our deployed software is healthy and has not degraded. We can use the insights from these tools in our development teams to improve the quality of the software on a frequent, ongoing basis.

Infrastructure Automation

For software targeting Windows and .NET, infrastructure is no longer the bottleneck it once was. VMs can be built and provisioned within minutes, whether we're using on-premise Hyper-V or VMware, or commodity public cloud such as AWS or Azure. However, Continuous Delivery needs repeatability of infrastructure configuration, which means automation, not point-and-click or manually executed PowerShell commands.

For Windows and .NET, our future-proof infrastructure or platform choices are currently:

IaaS

VMs running on commodity cloud

PaaS

Effectively just Microsoft Azure, using features such as [Azure Cloud Services](#)

In the future, we will see containerization technologies like [Docker for running Windows-specific workloads](#), but these remain Linux-only at the time of writing. [Microsoft is working in partnership with Docker](#) to extend the Docker API to support containers running Windows.

NOTE

Most organizations that need pools of VMs for testing should use commodity cloud providers like [Amazon AWS](#), [Microsoft Azure](#), [ElasticHosts](#), or [Rackspace](#). The real costs of building and maintaining a self-hosted VM infrastructure (sometimes called “private cloud”) are much higher than most people realize, because many costs (power, cooling, resilience, training, disaster recovery) are hidden. Unless your organization has niche requirements (such as high throughput, low latency, or compliance restrictions), you should plan to use commodity cloud infrastructure.

Shared Versus Dedicated Infrastructure

For Continuous Delivery to be effective, we need to minimize any friction caused by shared infrastructure. If several product teams are regularly waiting for another team to finish using a testing environment, it becomes difficult to maintain regular, reliable releases.

Some infrastructure and tools are worth sharing so that we have a single source of truth: the version control system, the artifact repository, and some monitoring and metrics tools. Most other tools and technologies should be aligned and dedicated to a specific group of products or services. Dedicated infrastructure costs more in terms of hardware, but usually saves a significant amount of money in time wasted waiting for environments or retesting after another team broke a shared environment.

TIP

An effective pattern for shared environments is to limit the testing time to 20 or 30 minutes, including deployments; if the new features deployed by one team do not pass automated checks in that time window, the code is automatically rolled back and the team must fix the problems before trying again.

Avoid deploying applications and services from multiple teams onto the same virtual or physical hardware. This “deployment multi-tenancy” leads to conflicts between teams over deployment risks and reconfiguration. Specifically, a team running its software on a set of Windows machines should be able to choose to run `iisreset` at any time, knowing that they will affect only their own services (see [Figures 6-1](#) and [6-2](#)).

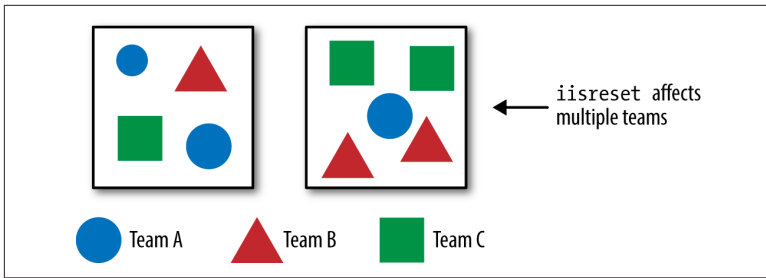


Figure 6-1. Deployment multitenancy—*iisreset* causes friction

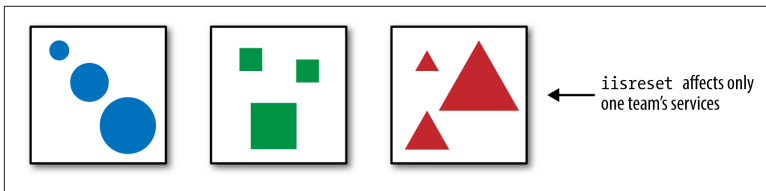


Figure 6-2. Deployment single tenancy—*iisreset* affects just one team

Using a Test-First Approach to Infrastructure

Automation is of little use if we cannot test the resulting infrastructure before we use it. Test-first approaches to software (such as **TDD**) have worked well for application software, and we should use similar test-first approaches for infrastructure code too.

To do this effectively with Windows and .NET, we can use **Packer** and **Boxstarter** for building base images; **Vagrant** for VM development; **ServerSpec** for declarative server testing; a configuration management tool like **Chef**, **Puppet**, **Ansible**, and/or **PowerShell DSC**; and a CI server with remote agents (such as **TeamCity** or **GoCD**).

Use **Packer**, **Boxstarter**, and **ISO files** to create base Windows images, which we can then use in a traditional TDD-style coding loop:

1. Write a failing server test (such as expecting a .NET application to be deployed in IIS on the VM).
2. Implement enough logic to make the test pass (perhaps deploy the application using MSDeploy).
3. Refactor the code to be cleaner and more maintainable.

4. Push the configuration changes to version control.
5. Wait for the CI system to
 - a. Pull the changes from version control,
 - b. Build the changes using Vagrant and ServerSpec, and
 - c. Show that our tests have passed (a “green” build).
6. Repeat.

By building and testing our VM images and configuration management scripts using TDD and CI, we help to ensure that all changes to servers are tracked and derived from version control (Figure 6-3). This reduces the problems that beset many Windows server environments, caused by manual or opaque SCCM-driven configuration.

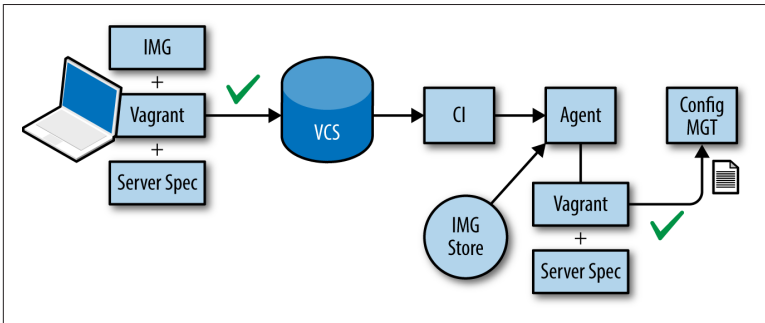


Figure 6-3. Use the base images to follow a TDD-style cycle



Some organizations and teams using Windows and .NET have tried to use raw PowerShell to automate the configuration and/or provisioning of infrastructure. Almost invariably, these PowerShell-only attempts seem to progress well for a few months until they reach a certain size and complexity, when they begin to fail. The lack of tests begins to become a major problem as the code becomes unwieldy and brittle, resembling a poorly written version of Chef or Puppet! A better approach is to build on a foundation of Chef/Puppet/Ansible/DSC.

JustGiving: Infrastructure Automation for Continuous Delivery

JustGiving is the world's largest fundraising platform. During 2014 alone, JustGiving has helped tens of millions of people raise over \$700 million.

Founded in 2000, JustGiving's platform is based substantially on Windows and .NET technologies, with some Linux-based auxiliary services. In 2013, the company began a program of modernizing its 13-year-old codebase and infrastructure, and has adopted an architecture of microservices running on AWS.

The JustGiving technology teams use GitHub for version control (with GitHub Flow as the branching/review model), JIRA for issue tracking, TeamCity for builds and Continuous Integration, Artifactory for deployable artifacts, and GoCD for Continuous Delivery deployment pipelines.

Crucially, by using Chef for server configuration management, the JustGiving teams treat infrastructure as code, and all infrastructure changes use the GitHub/TeamCity/GoCD build and deployment process to test and roll out changes, just like application code. This allows the teams to spin up and shut down AWS environments on demand, including entire copies of the Production environment.

Owain Perry, Software Architect at JustGiving, says:

We're very happy with our Windows-on-AWS architecture. Although Windows historically presented some challenges compared to Linux—particularly around software updates—we've found that modern .NET-friendly tooling like NuGet, Chocolatey, GoCD, Artifactory, and TeamCity help us to make changes quickly and safely. Ultimately, this means we can scale up our

systems rapidly in times of high demand, particularly when a charity campaign goes viral.

The combination of automated build and deployment and automated infrastructure configuration is enhanced with powerful log aggregation and search capabilities provided by an in-house Elastic-Search/LogStash/Kibana (ELK) system, allowing searching and alerting across all application and infrastructure components.

Patching and OS Updates

Our approach to OS patching and Windows Updates needs to be compatible with Continuous Delivery. For Windows and .NET, this means that we need to reconsider how tools like **SCCM**, **WSUS**, and **WUFB** can interact with servers in our infrastructure, particularly when we have Chef/Puppet/Ansible/DSC controlling aspects of server configuration.

In practice, whether we use commodity cloud or self-hosted infrastructure, this means following the pattern set by commodity cloud providers like AWS and Azure:

- Server images are provided prepatched to a known patch level.
- Product teams regularly update their base images to newer versions, retesting their code.
- Software and infrastructure is designed to cope with servers being rebooted for patching during the day (for zero-day vulnerabilities).

It is crucial to establish a boundary between the configuration undertaken by Chef/Puppet/Ansible/DSC and the configuration undertaken by SCCM/WSUS/WUFB. Several organizations have had success with letting SCCM/WSUS handle OS-level patches and updates, leaving all other updates and patches (such as for SQL Server) to the scriptable configuration tool. The goal is to avoid the fights that occur with a split-brain configuration approach.

Summary

For Continuous Delivery, we need to treat infrastructure as code, using proven techniques like TDD. We should expect to take advantage of existing tools for infrastructure automation and configuration rather than “rolling our own,” and we need to consider the possible negative effects of shared servers.

The Tricky Bits of Continuous Delivery

In the previous chapters, we have covered some approaches for solving some of the technical challenges of achieving Continuous Delivery with Windows and .NET. However, tooling alone will not result in Continuous Delivery for your software. Continuous Delivery also requires changes to your organization and practices, and we explore these more tricky aspects in this chapter.

Organizational Changes

Organizations can make Continuous Delivery easier by making some changes to the shape, size, and responsibilities of teams. One of the most effective changes is to *align software delivery teams with a product or service*: the team works on a program of changes and features for one or more product and service, but only that team. Instead of projects we use programs of work that have a clear and consistent timeline over many months and years.

This alignment with products and services typically helps to work around the problems of **Conway's law**—the tendency to push the software we build into shapes that resemble the communication structures in the organization. This alignment with services works well with recent practices such as **DevOps**, where responsibility for the build and operation of software systems is shared between several teams with complementary skills, as shown in Figures 7-1 and 7-2.

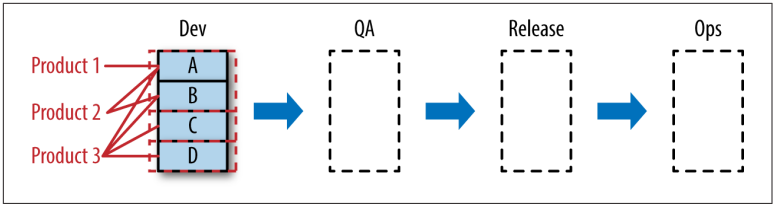


Figure 7-1. Function-focused teams produce a need for handoffs

Product 1	Team A	Dev	Tester	Release	Ops
Product 2	Team B	D	T	R	O
Product 3	Team C	D	T	R	O

Figure 7-2. Cross-functional product-focused teams

To aid Continuous Delivery, development tasks need to be approached in a manner that promotes continuous flow and short feedback loops. There are various agile methodologies aimed toward these goals such as Scrum, Kanban, and XP. Any of these methodologies may work for your team—it is something you will need to experiment with. The key factors are to keep work in progress (WIP) limits low, highlight any blocked tasks, and fully complete tasks to “release” before moving on to another task. It is essential to keep the work “flowing.” The Theory of Constraints, as described in *The Goal* [GoldrattCox] and more recently *The Phoenix Project* [Kim], is an approach for highlighting bottlenecks in the development process and as such, in preventing a buildup of incomplete changes from becoming stalled.

Organizations should expect to allocate the equivalent of around one person on build and deployment activities per development team of 8–10 people (what Amazon has referred to as a *two-pizza team*). Build and deployment are so fundamental to Continuous Delivery that they need strong, ongoing investment if Continuous Delivery is to be successful.

Testing for Continuous Delivery looks very different from traditional approaches. Instead of separate QA or Release Testing teams, we embed testers within product/service teams, so that the valuable questioning mindset of the tester can complement the optimistic mindset of the developer, and we **build quality in**.

Architectural Changes (SOA/Microservices)

The fast rate of release in a Continuous Delivery environment promotes certain architectural approaches of a platform. Releasing often, with small changes, is extremely difficult to do with a monolithic architecture.

We recommend rearranging the architecture into smaller, focused, domain-specific applications—a service-oriented architecture (SOA) or even one composed of microservices. These architectures come with their own challenges: data consistency, service availability, and partition tolerance (known as the CAP theorem). The benefit of a distributed system is that it allows for each of the components to change and evolve with only loose coupling between them.

If the contracts between each component are well defined and stable, this approach allows for individual components to be developing and releasing change at the same time. The overall rate of change will greatly increase over that of a monolith where change has to be highly coordinated.



TIP

Read Up on Distributed Systems Theory

As you are developing a distributed system it is worthwhile reading up on some of the theory and learning from the experience of other companies using these techniques: CAP theorem, fault tolerance, high availability, correlation IDs for transaction tracing, Netflix's Simian Army, and so on.

Ancestry.com: Software Architecture Changes for Continuous Delivery

Ancestry.com is the world's largest online resource for family history with more than two million paying subscribers across all its sites. Ancestry.com began its extensive and unique collection of billions of historical records around 1996 and is now the worldwide leader in online family history research.

Between 2010 and 2012, the number of Ancestry.com subscribers doubled (from 1 million to 2 million), along with the number of page views (25 million per day to 50 million per day). The existing software systems, built on Windows and .NET, were becoming increasingly difficult to change and release; a software architecture designed for 100,000 users was showing strain at approaching two million users, and the new users were demanding more new features, more quickly.

Almost every part of the system was tightly coupled to another part, resulting in a “big blob” monolithic system that was difficult to change, test, and release. The Ancestry.com teams knew that to move faster, they needed to adopt Continuous Delivery practices, with small, incremental changes flowing regularly toward Production. Crucially, the teams realized that Continuous Delivery required a major change in the Ancestry.com software and systems architecture and began “adopting product architectures that permit work to flow in small, decoupled batches” [Reinertsen]. Ancestry.com began creating a *software architecture designed for agility*.

Russ Barnet, Chief Architect at Ancestry.com, explains that:

What we learnt is that software architecture affects agility and Continuous Delivery capability as much or more than other factors. Process and tool improvements alone are insufficient; good architecture techniques enable effective Continuous Delivery at large scale.

John Esser, Director of Engineering Productivity at Ancestry.com, adds:

By re-architecting for Continuous Delivery, and using tools like Chef and GoCD in combination with Windows and .NET, we were able to move from infrequent, difficult code deployments to weekly, daily, and even hourly deployments, whilst improving our availability and mean time to recovery.

Implemented on the Windows/.NET platform, the new software architecture had three key architectural principles:

- Many small, single-responsibility components (rather than fewer components with multiple responsibilities)
- Components that could be separately deployable
- Components that could be rolled back independently, with forward and backward compatibility

The 40 tech teams at Ancestry.com also addressed key operational challenges around fault detection and recoverability by exposing the runtime status of each component so that deep-level checks could reveal exactly where faults had occurred. Combined with explicit service-level agreement (SLA) checks on inbound *and* outbound API calls, these changes helped significantly improve the operability of the Ancestry.com systems.

Operational Features

Many organizations treat functional and nonfunctional requirements separately, with nonfunctional aspects often deprioritized or given to IT Operations teams to handle. This approach is not suitable for Continuous Delivery, because Continuous Delivery needs high-quality, well-tested, production-ready software, not just a set of user-visible features that seem to work on a developer's laptop or in a UAT (User Acceptance Test[ing]) environment.

There are many invisible considerations that we need to address for Continuous Delivery, including performance, resilience, deployability, security, and availability. The ability to test new code—testability—becomes an important consideration for Continuous Delivery; we generally prefer code that is more easily testable than code that is, say, more clever but less testable.

There are several useful techniques for addressing operational concerns. Rather than separating functional and nonfunctional requirements, organizations should prioritize these requirements together in a single backlog, pairing up a Product Owner and a Tech Lead if necessary. We can also identify an IT Operations role as a secondary user type (or persona) and write User Stories for them: “As an IT Ops person, I need an automated way to check the runtime status of a component so that I can diagnose health problems within 60 sec-

onds,” for instance. Some organizations place a kind of “tax” on product budgets—usually around 20%–30% of the total product budget—which is used for operational concerns. The tax is essentially an honest way of accounting for the operational costs incurred with modern, complex software systems; either we pay for them up front with a slice of the product budget, or the organization pays for them in unplanned downtime, security breaches, or rework after failed technical testing.

Summary

Continuous Delivery requires changes not only to the technologies and techniques we use for building and deploying software but also to the way in which we structure our teams, our software architecture, and our approach to operational concerns. These tricky aspects are essential for a sustainable approach to Continuous Delivery that will provide lasting results regardless of the tooling and technologies used.

Bibliography

- [AmblerSadalage] Scott Ambler and Pramod Sadalage. *Refactoring Databases* (Addison-Wesley, 2006).
- [GoldrattCox] Eliyahu M. Goldratt and Jeff Cox. *The Goal: A Process of Ongoing Improvement* (North River Press, 1984).
- [Hammer] Derek Hammer. “TFS is destroying your development capacity.” Blog post: <http://www.derekhammer.com/2011/09/11/tfs-is-destroying-your-development-capacity.html>, 2011.
- [Hendrickson] Elisabeth Hendrickson. *Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing* (Pragmatic Bookshelf, 2013).
- [HumbleFarley] Jez Humble and Dave Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* (Addison-Wesley, 2010).
- [Kim] Gene Kim, Kevin Behr, and George Spafford. *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win* (IT Revolution Press, 2013).
- [Laycock] Rachel Laycock. “Continuous Delivery on Windows.” Articles in ThoughtWorks P2 magazine: <http://thoughtworks.github.io/p2/issue06/cd-in-windows-part-1/>, 2013–2014.
- [MannsRising] Mary Lynn Manns and Linda Rising. *Fearless Change* (Addison-Wesley, 2005).

- [Nygard] Michael Nygard. *Release It!: Design and Deploy Production-Ready Software* (Pragmatic Bookshelf, 2007).
- [Poppendieck] Mary Poppendieck and Tom Poppendieck. *Implementing Lean Software Development: From Concept to Cash* (Addison-Wesley, 2006).
- [Reinertsen] Don Reinertsen. *Principles of Product Development Flow* (Celeritas Publishing, 2009).
- [SmithSkelton] Steve Smith and Matthew Skelton (eds). *Build Quality In* (LeanPub, 2014).

Case Studies

7digital

“7digital: Continuous Integration for Continuous Delivery” on page 24

Ancestry.com

“Ancestry.com: Software Architecture Changes for Continuous Delivery” on page 58

Carnect

“Carnect: Version Control Changes for Continuous Delivery” on page 8

JUST EAT

“JUST EAT: Utilizing Metrics for Successful Continuous Delivery” on page 45

JustGiving

“JustGiving: Infrastructure Automation for Continuous Delivery” on page 51

LateRooms

“LateRooms: Deployment Pipelines for Continuous Delivery” on page 31

About the Authors

Chris O'Dell has developed software with Microsoft technologies for over 10 years. Between 2012 and 2014, she led the API team at 7digital, a leading provider of music streaming services worldwide; she currently works on the platform engineering team at JUST EAT in London. In all of her roles she has promoted practices we now know as Continuous Delivery, including TDD, version control, and Continuous Integration. Chris is a contributor to the book *Build Quality In* [SmithSkelton].

Matthew Skelton has been developing software on Microsoft Windows since 1998, and has consistently pushed quality and repeatability (version control, testability, logging for traceability, etc.) within every company he has worked at. Between 2011 and 2014, he was Build and Deployment Architect at Trainline, the UK's busiest travel booking site, where he led the transition from manual deployments to per-team deployment pipelines for Continuous Delivery. Matthew is Principal Consultant at Skelton Thatcher Consulting and coeditor of (and a contributor to) the book *Build Quality In* [Smith-Skelton].