Community Experience Distilled

Continuous Delivery and DevOps – A Quickstart Guide

Second Edition

Deliver quality software regularly and painlessly by adopting CD and DevOps

Paul Swartout

[PACKT] PUBLISHING open source* community experience distilled

Continuous Delivery and DevOps – A Quickstart Guide

Second Edition

Deliver quality software regularly and painlessly by adopting CD and DevOps

Paul Swartout

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Continuous Delivery and DevOps – A Quickstart Guide

Second Edition

First published: November 2012

Second edition: December 2014

Production reference: 1191214

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78439-931-3

www.packtpub.com

Credits

Author
Paul Swartout

Reviewers
Max Manders
Adam Strawson
Diego Woitasen

Commissioning Editor
Julian Ursell

Acquisition Editor
Reshma Raman

Content Development Editor
Anand Singh

Technical Editor
Ryan Kochery

Copy Editors
Janbal Dharmaraj
Sayanee Mukherjee
Karuna Narayanan

Project Coordinator
Rashi Khivansara

Proofreaders
Simran Bhogal
Maria Gould
Ameesha Green
Paul Hindle
Joanna McMahon

Indexer
Rekha Nair

Graphics
Disha Haria

Production Coordinator
Manu Joseph

Cover Work
Manu Joseph

About the Author

Paul Swartout has spent over 20 years working in the IT industry. Starting out as a developer with a small software house, he has filled a number of roles over the years, including software engineer, system administrator, project manager, program manager, operations manager, scrum master, Agile coach, and software development manager. He has worked across a number of different industries and sectors—from supply chain through manufacturing, education, and retail to entertainment—and within organizations of various sizes, from start-ups to multinational corporates.

He is passionate about software and how it is delivered. Since he first encountered Agile over a decade ago, he has been committed to the adoption and implementation of Agile techniques and approaches to improve the efficiency, output, and lives of everyone involved in software development.

Over the past few years, he has been heavily involved in the CD and DevOps movement, from heading the team within Nokia that implemented said ways of working to blogging, presenting, authoring, and evangelizing to whoever is in earshot. He strongly believes that CD and DevOps add massive value to the way software is delivered, and he wants to ensure as many people realize this as possible.

Paul lives in a small seaside town in the southwest of the UK with his wife, daughters, and two small yapping things.

He is a software development manager and Agile coach working for Microsoft, based in the MixRadio team in Bristol in the UK.

He has also worked on *Continuous Delivery and DevOps: A Quickstart Guide*.

You can contact Paul and find out what he's up to via `www.swartout.co.uk`.

Acknowledgments

Firstly, I would like to say a big thank you to my darling wife, Jane, who has yet again had to put up with a husband, who, for the past few months, has done little more than spend every spare moment staring at a computer screen typing things, frowning, then typing more things—things that eventually turned into this book.

Next is my good friend John Clapham, whose level-headed approach and consistent vision helped make the implementation of CD and DevOps within MixRadio the success it was. Without that success, there would be little to write about.

A big thank you to John Fisher for allowing me to include his transition curve within the book again.

Thank you to everyone who purchased and read the first edition—without you, the opportunity for this second edition would never have come to pass.

Lastly, I want to thank the global CD and DevOps community for their never-ending commitment, passion, enthusiasm, and evangelism to bring this amazing way of working to the masses. Keep up the good work.

About the Reviewers

Max Manders is a recovering PHP web developer and former sysadmin, who currently works as a systems developer and ops engineer, helping to run the Operations Center for Cloudreach, an Amazon Web Services Premier Consulting Partner. He has put his past experiences and skills to good use to evangelize all things DevOps, working to master Ruby and advocating infrastructure-as-code as a Chef practitioner.

He is a cofounder and organizer of Whisky Web, a Scottish conference for the web development and ops community. When he's not writing code or tinkering with the latest and greatest monitoring and operations tools, he enjoys the odd whisky and playing jazz and funk trombone. He lives in Edinburgh with his wife, Jo, and their cats, Ziggy and Maggie.

It's been an absolute pleasure to have the opportunity to provide a technical review of this book. I hope you enjoy reading it as much as I did! We worked to quite an aggressive schedule, so I'd like to thank Jo for being so understanding while I buried my head in my Mac (not that unusual though!). Also, a big shout out to my amazing colleagues' ongoing support, especially those who remind me that everything is awesome even when things get hectic!

Adam Strawson is an engineer with 8 years of experience in the industry, with experience ranging from web development to system administration. He has worked in a number of fields, from agency to in-house, including financial marketplaces and SASS products. He has been involved in the DevOps community for a number of years and has introduced the process, including CI and CD, in his previous and current positions.

He currently lives in the seaside town of Brighton, UK, and works as a software engineer for The Student Room Group. He can be contacted on Twitter at `@adamstrawson`, or on his blog at `http://adamstrawson.com`.

Diego Woitasen has more than 10 years of experience in Linux and the open source consulting industry. Along with Luis Vinay, he is the cofounder of flugel.it. As self-denominated infrastructure developers, they apply all their years of experience helping all sorts of companies and new movements related to interdisciplinary cooperative working environments to embrace the DevOps culture.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Continuous Delivery (CD) and DevOps is fast becoming the next big thing in relation to the delivery and support of software. Strictly speaking, that should read *the next big things*, as CD and DevOps are actually two complementary yet separate approaches:

- Continuous Delivery, as the name suggests, is a way of working whereby quality products, normally software assets, can be built, tested and shipped in quick succession—thus delivering value much sooner than traditional approaches
- DevOps is a way of working whereby developers and IT system operators work closely, collaboratively, and in harmony towards a common goal with little or no organizational barriers or boundaries between them

This book will provide you with some insight into how these approaches can help you optimize, streamline, and improve the way you work and, ultimately, how you ship quality software. Included in this book are some tricks and tips based on real-world experiences and observations; they can help you reduce the time and effort needed to implement and adopt CD and DevOps, which, in turn, can help you reduce the time and effort required to consistently ship quality software.

What this book covers

Chapter 1, *Evolution of a Software House*, introduces you to ACME systems and the evolution of their business from a fledgling start-up through the growing pains following acquisition by a global corporation, to the best of both worlds.

Chapter 2, *No Pain, No Gain*, introduces techniques that can be used to determine the current pain points within your software delivery process and they stem from.

Chapter 3, *Plan of Attack*, gives you some pointers on how the success of implementing CD and DevOps can be defined and how this success can be measured.

Chapter 4, *Culture and Behaviors*, highlights the importance of the "human" factors that must be taken into account if you want CD and DevOps to succeed.

Chapter 5, *Approaches, Tools, and Techniques*, will give you some options around the various tools and techniques (some technical, some not so) that can help with the implementation and adoption of CD and DevOps.

Chapter 6, *Hurdles Along the Way*, will give you some useful tips and tricks to overcome or avoid the bumps in the road during the adoption of CD and DevOps.

Chapter 7, *Vital Measurements*, focuses on the various metrics and measures that can be used to monitor and communicate the relative success of CD and DevOps adoption.

Chapter 8, *Are We There Yet?*, focuses on the sorts of things you should be looking out for once the adoption of CD and DevOps has become embedded in your day-to-day ways of working.

Chapter 9, *The Future is Bright*, will provide some insight into how you can take CD and DevOps techniques and experience beyond the traditional software delivery process.

Appendix A, *Some Useful Information*, provides you with some more detailed information on the tools referenced within the book and some useful contacts within the global CD and DevOps community.

Appendix B, *Where Am I on the Evolutionary Scale?*, provides you with one simplistic way to determine how advanced your CD and DevOps adoption is.

Appendix C, *Retrospective Games,* provides example agile games that can be used in conjunction with the techniques covered in *Chapter 2*, *No Pain, No Gain*.

Appendix D, *Vital Measurements Expanded*, provides some additional background on and advancement of the areas covered in *Chapter 7*, *Vital Measurements*.

What you need for this book

There are many tools mentioned within the book that will help you no end. These include technical tools such as Jenkins, GIT, Docker, Vagrant, IRC, Sonar, and Graphite, and nontechnical tools and techniques such as Scrum, Kanban, agile, and TDD.

You might have some of these (or similar) tools in place, or you might be looking at implementing them, which will help. However, the only thing you'll really need to enjoy and appreciate this book is the ability to read and an open mind.

Who this book is for

Whether you are a software developer, IT system administrator/operator, project manager, or CTO, you will have a common problem: regularly shipping quality software is painful. It needn't be.

This book is not focused on a specific demographic or specific type of person. If you've never heard of CD or DevOps, this book will give you an insight into what all the fuss is about. If you have already set out to adopt CD and/or DevOps, then this book can help by providing some useful tips and tricks. If you know everything there is to know about both/either subject, then this book will help reaffirm your choices and might provide some additional things to chew over.

All in all, the target audience is quite broad: anyone who wants to understand how to painlessly and regularly ship quality software.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

New terms and **important words** are shown in bold.

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: `https://www.packtpub.com/sites/default/files/downloads/9313OS_Graphics.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Evolution of a Software House

As described in the *Preface*, both **Continuous Delivery** (**CD**) and **DevOps** are complementary ways of working. The former assists with shipping quality software quickly, the latter helps harmonize the teams that deliver and support said software. Both approaches can help you to optimize, streamline, and improve the way you work. Ultimately, both will help you ship quality software.

Before we get onto the meat of CD and DevOps, let me introduce you to **ACME systems**—a typical software business—and walk you through their trials, tribulations, and evolution. The topics we will cover in this chapter are as follows:

- How ACME systems started from humble beginnings
- The growing pains it went through to become successful
- The positives and negatives that came from success and dramatic growth
- The advantages that came with adopting CD and DevOps ways of working
- How it adapted to utilize what it had learned to drive their business into new markets and opportunities

Without further ado, let's meet ACME systems.

A brief history of ACME systems

This fictional software business started out—as many successful tech companies do—in the garage of one of the founders. The founders were visionaries with big ambitions, good ideas, and a little bit of cash.

After a few years of hard work, determination, much blood, sweat, and tears, the dreams of the founders were realized. The business is recognized as a leader in its field and is then acquired by a multinational corporate. This acquisition brings with it the funding and resources needed to allow the business to grow and expand to become a global player. However, with corporate owners comes corporate responsibilities, rules, bureaucracy, and processes.

The ACME systems team start to find it increasingly difficult and painful to deliver quality software. They adopt and adhere to the parent company's processes to improve quality and reduce risk, but this makes the seemingly simple task of delivering software, laborious and extremely complex.

They come to an evolutionary crossroad and have to make a decision either to live with the corporate baggage that they have inherited and potentially face extinction, or try and get back to the good old days and good old ways that had reaped rewards previously.

While trying to decide which way to go, they discover they have another choice —implement CD and DevOps—which could give them the best of both worlds. As luck would have it that is exactly what they did.

Over the next few pages, we'll go through this evolution in a little more detail. As we do, you may recognize some familiar traits and challenges.

The name ACME is purely fictional and based upon the ACME Corporation, first used in *Road Runner Cartoons* in the 1950s—just in case you were wondering.

We'll start with the initial incarnation of the ACME systems business, which for want of a better name, will be called ACME systems version 1.0.

ACME systems version 1.0

Some of you have most probably worked for (or currently work for) a small software business. There are many such businesses scattered around the globe and they all have one thing in common—they need to move fast to survive and they need to entice and retain customers at all costs. They do this by delivering what the customer wants just before the customer needs it. Deliver too soon and you may have wasted money on building solutions that the customer decides they no longer need, as their priorities or simply their minds have changed. Deliver too late and someone else may well have taken your customer—and more importantly, your revenue—away from you. The important keyword here is *deliver*.

As mentioned earlier, ACME systems started out in humble beginnings; the founders had a big vision and could see a gap in the market for a web-based solution. They had an entrepreneurial spirit and managed to attract backers who injected the lifeblood of all small businesses—cash.

They then went about sourcing some local, keen, and talented engineers and set about building the web-based solution that bridged the gap in the market, which they had seen before anyone else could.

At first, the going was slow and the work was hard; a lot of pre-sales prototypes needed to be built in a hurry—most of which never saw the light of day—some went straight into production. After many long days, nights, weeks, and weekends, things started to come together. Their customer base started to grow and the orders started rolling in; as did the revenue. Soon the number of employees was in double figures and the founders had become directors.

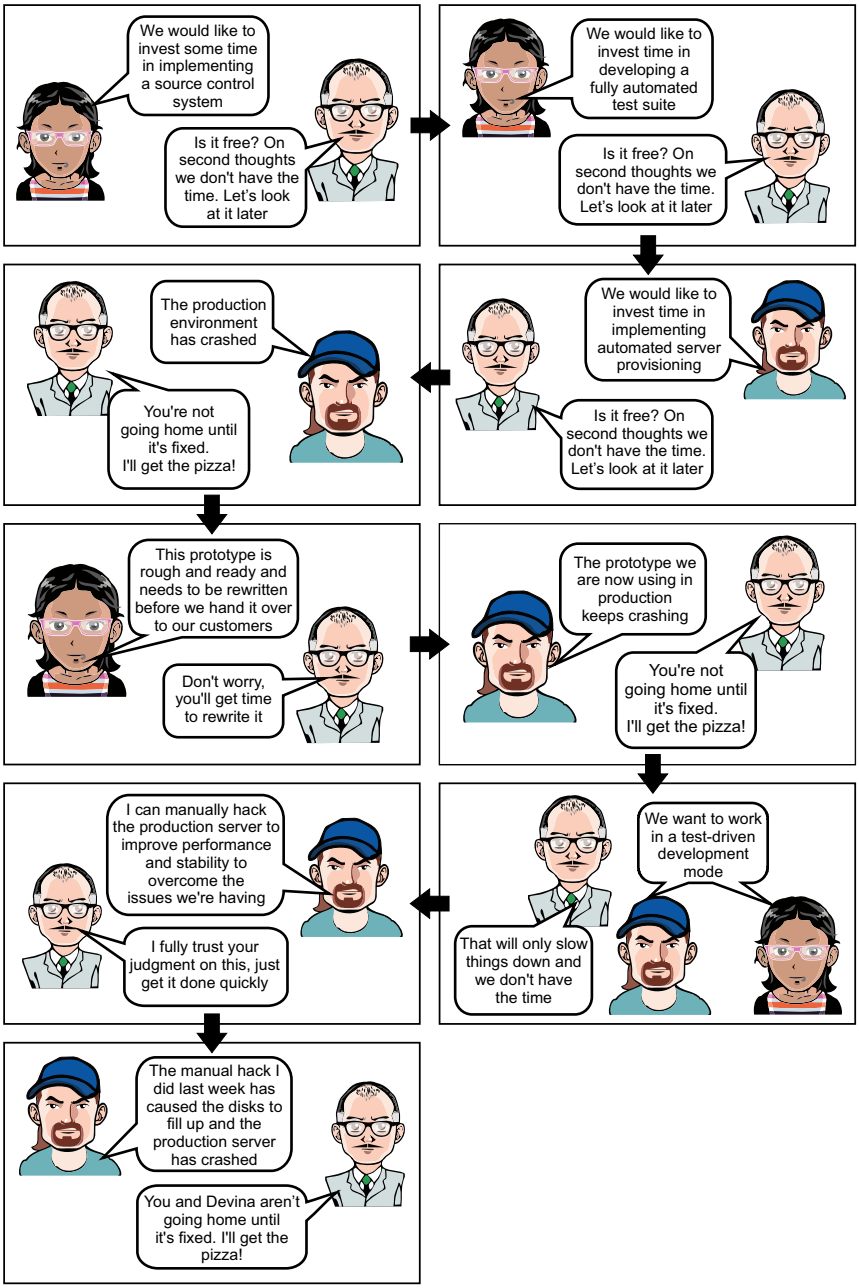
So, I hear you ask, "What has this got to do with CD or DevOps?" Well, everything really. The culture, default behaviors, and engineering practices of a small software house are what would be classed as pretty good in terms of CD and DevOps. For example:

- There are next to no barriers between developers and operations teams—in fact, they are generally one and the same
- Developers normally have full access to the production environment and can closely monitor their software
- All areas of the business are focused on the same thing, that being to get software into the production environment as quickly as possible and thus delight customers
- Speed of delivery is of the essence
- When things break, everyone swarms around to help fix the problem—even out of hours
- The software evolves quickly and features are added in incremental chunks
- The ways of working are normally very agile

There is a reason for stating that the culture, default behaviors, and engineering practices of a small software house would be classed as *pretty good* rather than *ideal*. This is because there are many flaws in the way a small software house typically has to operate to stay alive; for example:

- Corners will be cut to hit deadlines, which compromises software design and elegance
- Application security best practice is given short shrift or even ignored
- Engineering best practices are compromised to hit deadlines
- The concept of technical debt is pretty much ignored
- Testing is not in the forefront of the developer's mind and even if it were, there may not be enough time to work in a test-driven development way
- Source and version control systems are not used religiously
- With unrestricted access to the production environment, tweaks and changes can be made to the infrastructure with little or no audit trail
- Software releasing will be mainly manual and most of the time an afterthought of the overall system design
- At times, a rough and ready prototype may well become production code without the opportunity for refactoring
- Documentation is scant or nonexistent—that which does exist is most probably out of date
- The work-life balance for an engineer working within a small software house is not sustainable and *burn out* does happen

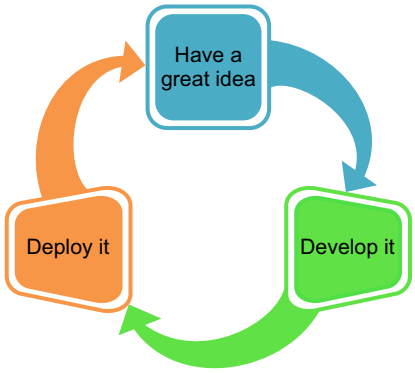
To emphasize this, let's have a look at a selection of typical conversations between three individuals within the ACME systems team: Stan, the manager; Devina, the developer; and Oscar, the operations guy.



We'll now have a look at the software delivery process for ACME systems version 1.0, which, to be honest, shouldn't take too long.

Software delivery process flow version 1.0

The following diagram gives an overview of the simple process used by ACME systems to deliver software. It's simple, elegant (in a rough-and-ready kind of way), and easy to communicate and understand.



An overview of the ACME systems version 1.0 software delivery process

Let's move forward a few years and see how ACME systems is doing and gain some insight into the benefits and pitfalls of being the leader in the field.

ACME systems version 2.0

The business has grown in size and turnover. The customer base is now global and the ACME systems software platform is being used by millions of customers on a daily basis. ACME systems is well established, well renowned, and recognized as being at the forefront in its area of expertise.

So much so that the board of ACME systems is approached by a multinational corporation and discussions are entered into regarding an acquisition. These discussions don't take long and the acquisition is completed within weeks. The board members are extremely happy, and the business as a whole sees this as a positive recognition that they have at last reached the big time.

At first, everything is good; everything is great in fact. The ACME systems team now has the backing they need to invest in the business and be able to scale out and obtain a truly global reach. They can also focus on the important things such as building quality software; scaling out the software platform; and investing in new technologies, tools, and R&D. The drier side of business—administration, program, project management, sales, marketing, and so on—can be passed to the new parent company that has all of this in place already.

The ACME systems team moves forward in excited expectation. The level of investment is such that the software engineering team doubles in size in a matter of months. The R&D team—as they're now called—introduces new tools and processes to enable speedy delivery of quality software. Scrum is adopted across the R&D team and the opportunity to fully exploit engineering best practices is realized. The original ACME systems platform starts to creak and is showing its age, so further investment is provided to re-architect and rewrite the software platform using the latest technologies. In short, the R&D team feels that it's all starting to come together and they have the opportunity to do it right.

In parallel to this, the ACME systems operations team is absorbed into the parent's global operations organization. On the face of it, this seems a very good idea; there are data centers filled with cutting-edge kit, global network capabilities, and scalable infrastructure. Everything that is needed to host and run the ACME systems platform is there. Like the R&D team, the operations team has more than they could have dreamed of. In addition to the tin and string, the operations team also has resources available to help maintain quality, control change to the platform, and ensure the platform is stable and available 24/7.

Sitting above all of this, the parent company also has well-established governance, program, and project management functions to control and coordinate the overall end-to-end product delivery schedule and process.

On the face of it, everything seems rosy and the teams are working more effectively than ever before. At first, this is true, but very soon, things start to take a downward turn. Under the surface, things are not that rosy at all.

We'll shift forward another year or so and see how things are:

- It is becoming increasingly difficult to ship software—what took days, now takes weeks or even months
- Releases are getting more and more complex as the new platform grows and more integrated features are added
- Despite the advances in re-architecting and rewriting the platform, there still remains large sections of legacy code deep within the bowels of the system, which refuse to die
- Developers are now far removed from the production environment and as such are ignorant as to how the software they are writing performs, once it eventually goes live
- There is a greater need to provide proof that software changes are of the highest quality and performance before they can go anywhere near the production servers

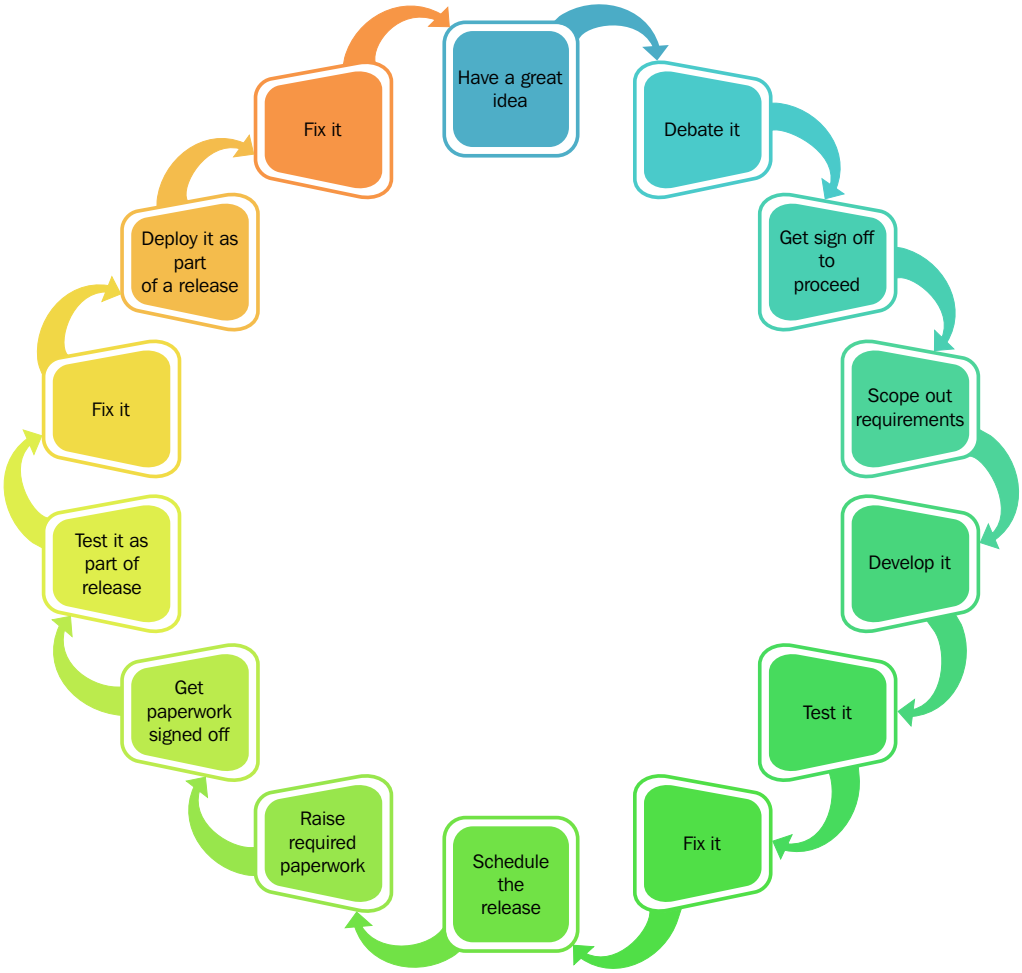
- Quality is starting to suffer as last minute changes and frantic bug fixes are being applied to fit into release cycles
- The technical debt amassed during the *fast and loose* days is starting to cause major issues
- Project scope is being cut at the last minute as features don't fit into the release cycles, which is leaving lots of redundant code lying around
- More and more development resources are being applied to assisting releases, which is impacting on the development of new features
- Deployments are causing system downtime—planned and unplanned
- Deadlines are being missed, stakeholders are being let down, and trust is being eroded
- The business's once glowing reputation is being tarnished

The main problem here, however, is that this attrition has been happening very slowly over a number of months and not everyone has noticed—they're all too busy trying to deliver.

Let's now revisit the process flow for delivering software and see what's changed—it's not a pretty picture.

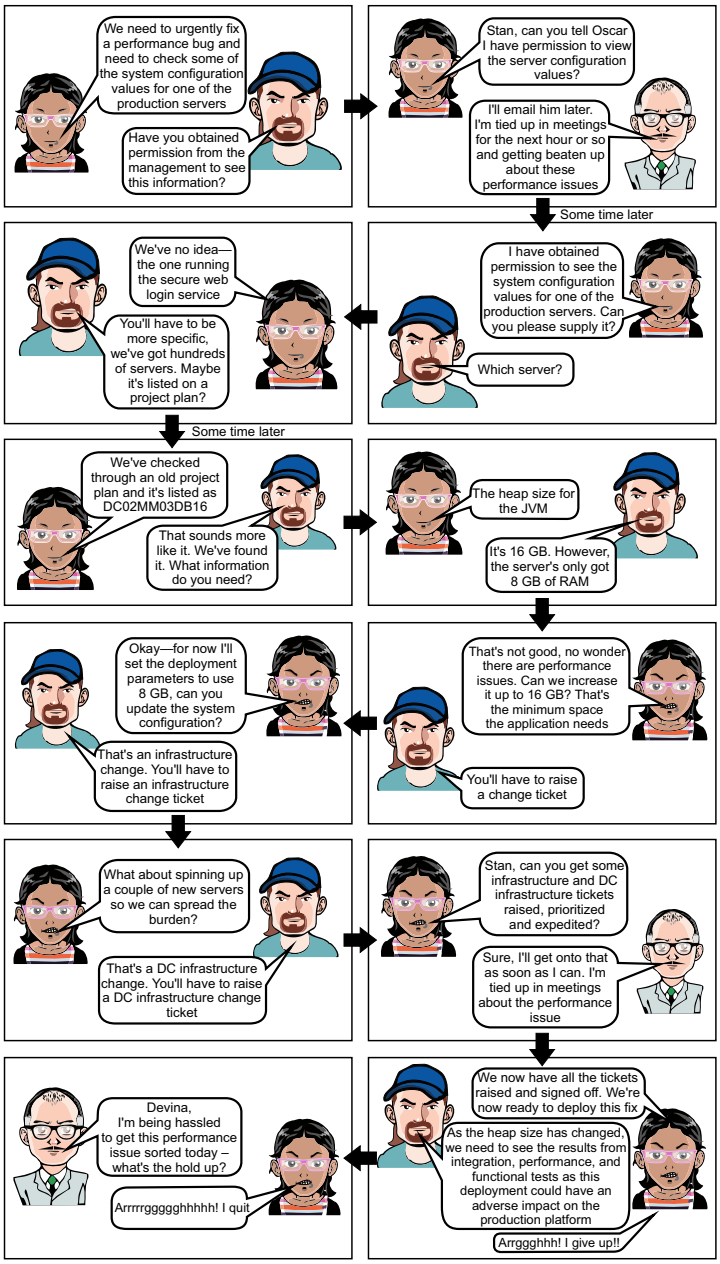
Software delivery process flow version 2.0

As you can see from the following diagram, things have become very complicated for the ACME systems team. What was simple and elegant has become complex, convoluted, and highly inefficient. The number of steps and barriers have increased, making it extremely difficult to get software delivered. In fact, it's increasingly difficult to get anything done. The following diagram gives you an overview of the ACME systems version 2.0 software delivery process:



An overview of the ACME systems version 2.0 software delivery process

Not only has the process become very inefficient—and to all intents and purposes broken—but the dialogue and the quality of the communication have also broken down. Let's again review a typical discussion between Devina, Oscar, and Stan regarding a live issue.



Okay, so this might be a little over the top, but it just serves to highlight the massive disjoint between the R&D and Operations team(s)—who you'll remember were pretty much one and the same in the early days of ACME systems. It should also be noted that this communication is now normally done via e-mail.

A few brave men and women

As was previously stated, not everyone noticed the attrition within the organization—luckily a few brave souls did. A small number of the ACME systems team are able to see the issues within the overall process as clear as day and they become determined to expose them and, more importantly, sort them out—it is just a question of how to do this while everyone is going at full pelt to get software delivered at all costs.

At first, they seek out a like-minded manager who has influence within the business and helps them to form a small virtual team. They then start identifying and breaking down the immediate issues and go about implementing the following tooling to ease some of the pain:

- Build and test automation
- **Continuous Integration** (**CI**)
- Automated deployment and monitoring solutions

This goes some way to address the issues but there are still some fundamental problems that tooling cannot address—the culture of the organization itself and the many disjointed silos within it. It becomes obvious that all the tools and tea in China will not bring pain relief; something more drastic is needed.

The team refocuses and works to highlight this now obvious fact to as many people as they can up and down the organization, while the influential manager works to obtain backing from the senior leadership to address it—which luckily is forthcoming.

We're now going on to the third stage of the evolution where things start to come back together and the ACME systems team regains their ability to deliver quality software when it is needed.

ACME systems version 3.0

The CD team—as they are now called—gets official backing from up high and becomes dedicated to addressing the problematic culture and behaviors, and developing ways to overcome and/or remove the barriers. They are no longer simply a technical team; they are a catalyst for change.

The remit is clear—do whatever is needed to streamline the process of software delivery and make it seamless and repeatable. In essence, implement what we now commonly refer to as CD and DevOps.

The first thing they do is to simply talk with as many people across the business as possible. If someone is involved in the process of getting software from conception to consumer and support it when it's live, they are someone you need to speak with. This not only gathers useful information but also gives the team the opportunity to evangelize and form a wider network of like-minded individuals.

The team has a vision, a purpose, and they passionately believe in what needs to be done, and have the energy and drive to do it.

Over the next few months, they embark on (among other things):

- Running various in-depth sessions to understand and map out the end-to-end product delivery process
- Refining and simplifying tooling based upon continuous feedback from those using it
- Addressing the complexity of managing dependencies and order of deployment
- Engaging experts in the field of CD to independently assess the progress being made (or not as the case may be)
- Arranging offsite CD training and encourage both R&D and Ops team members to attend the training together (it's amazing how much DevOps collaboration stems from a chat in the hotel bar)
- Reducing the many handover and decision-making points throughout the software release process
- Removing the barriers to allow developers to safely deploy their own software to the production platform
- Working with other business functions to gain trust and help them to refine and streamline their processes
- Working with R&D and operations teams to experiment with different agile methodologies such as **Kanban**

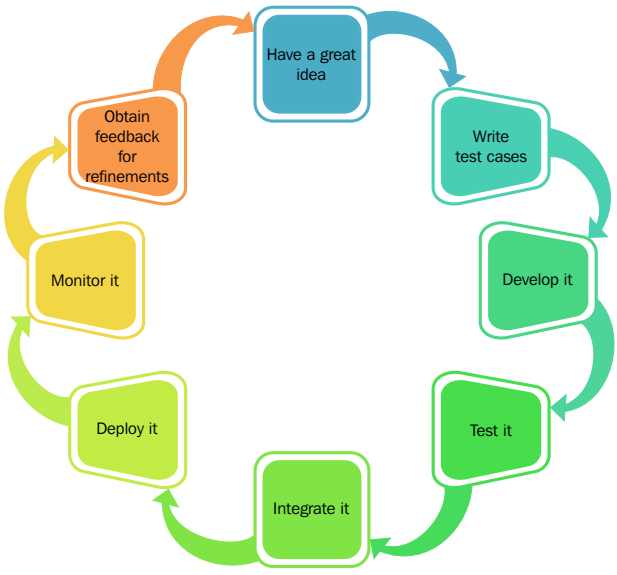
- Openly and transparently sharing information and data around deliveries and progress being made across all areas of the business
- Replacing the need for complex performance testing with the ability for developers to closely monitor their own software running in the production environment
- Evangelizing across all areas of the business to share and sell the overall vision and value of CD and DevOps

These initiatives are not easy to implement and it takes time to produce results but, after some months, the process of building and delivering software has transformed to the extent that a code change can be built, fully tested, and deployed to the production platform in minutes, many times per day—*all at the press of a button and initiated and monitored by the developer who made the change*.

Let's look again at the software delivery process flow to see what results have been realized.

Software delivery process flow version 3.0

As you can see from the diagram, the process looks much healthier. It's not as simple as version 1.0 but is efficient, reliable, and repeatable. Some much needed checks and balances have been retained from version 2.0 and optimized to enhance rather than impede the overall process.



An overview of the ACME systems version 3.0 software delivery process

This highly efficient process has freed up valuable DevOps resources so that they can focus on what they are best at—developing and delivering new software features and ensuring that the production platform is healthy and customers are again delighted.

The ACME systems team has gotten its mojo back and is moving forward with a new-found confidence and drive. They now have the best of both worlds and there's nothing stopping them.

ACME systems version 4.0

The ACME systems team have come through their challenges stronger and leaner but their story doesn't end there. As with any successful business, they don't rest on their laurels but decide to expand into new markets and opportunities—namely, to build and deliver mobile optimized clients to work with and complement their core web-based propositions.

With all they have learned throughout their evolution, they know they have an optimal way of working to allow them to deliver quality products that customers want, and they know how to deliver quickly and incrementally. However, the complexities of delivering code to a hosted web-based platform are not the same as the complexities of delivering code to an end consumer's mobile device—they are comparable but not the same. ACME systems also realizes that the process of delivering code to its production platform many times per day is under its control—code is being deployed to its infrastructure by its engineers using its tools—whereas it has little or no control over how its mobile clients are delivered, nor if and when the end consumer will install the latest and greatest version from the various app stores available. ACME systems also realizes that delivering a new version of its mobile clients many times per day is not viable nor welcome.

All is not lost—far from it. The ACME systems team has learned a vast amount throughout their evolutionary journey and decide to approach this new challenge as they did previously. They know they can build, test and deliver software with consistent quality. They know how to deliver change incrementally with little or no impact. They know how to support customers, and monitor and react quickly to change. They know their culture is mature and that the wider organization can work as one to overcome shared challenges. With this taken into account, here are a few of the things they decide to do:

- Agree on a realistic delivery cadence to allow for regular incremental changes without bombarding the end consumer
- Invest in new automated build, CI, and testing tools, which seamlessly integrate with and enhance the existing tooling

- Invest time and effort in nonfunctional features that will allow for greater visibility of what is running out in the wild, which again seamlessly integrates with the existing tooling and monitoring approach
- Ensure that the engineers delivering the mobile clients work closely with the backend engineers (DevOps) so that the client integrates seamlessly and doesn't cripple the existing production platform

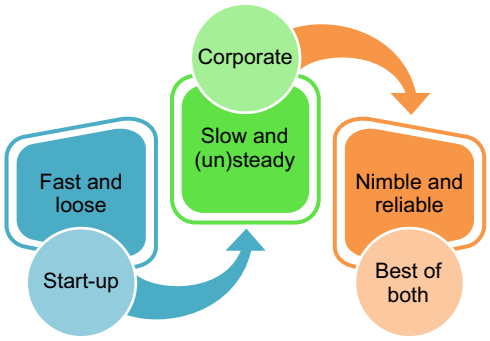
As the ACME systems team start to look into applying their established and proven approach to the new venture, they also discover another side effect of their newly rekindled success; they need to scale their platform and they need to do it as soon as possible. Given the timescales and urgency, the ACME systems team decides to move away from relying on their own datacenter and move towards a globally distributed "cloud-based" solution. This brings with it new challenges; the infrastructure is completely different, the provisioning tools are new, and the tools used to build and deliver software are incompatible with the existing ACME systems tools. Again, they take this in their stride and forge ahead with confidence using the ways of working, techniques and approaches that are now part of their DNA.

Could the ACME systems version 1.0 business have taken on these new challenges and succeeded? It's possible, but the results would have been mixed, the risks would have been much greater, and the quality much lower. It's pretty obvious that the ACME systems version 2.0 business would have had major struggles and by the time the products had hit the market, they would have been outdated and fighting for the market share with quicker and leaner competition.

If you would like to understand where you and your business sits within the CD and DevOps evolutionary scale, please see *Appendix B*, *Where Am I on the Evolutionary Scale?*

The evolution in a nutshell

Throughout this chapter, we have been following the evolution of ACME systems; where it started, the growing pains that came from success, how it discovered that being acquired brings with it negatives as well as positives, how it overcame its near extinction by adopting CD and DevOps, and how it regained its mojo and confidence to move forward. All of this can be represented by the following simple diagram:



An overview of ACME systems evolution

Summary

The ACME systems evolution story is not atypical of the many software businesses out there today. As stated previously, you may recognize and relate to some of the traits and challenges, and you should be able to plot where you, your business, or your employer currently sit within the stages detailed.

We'll now move from storytelling mode and start to look in more detail at some of the practical aspects of adopting CD and DevOps, starting with how one identifies the underlying problems that can—and do—stifle the delivery of quality software.

2
No Pain, No Gain

In the previous chapter, you were introduced to ACME systems and given an insight into how they realized that they had problems with their software delivery process (severely impacting their overall product-delivery capability), how they identified and addressed these problems, evolved, and after much hard work and some time, adopted a CD and DevOps ways of working. This isn't to say that you should simply dive in and adopt CD and DevOps because ACME systems did—far from it.

CD and DevOps, like any solution, can help you solve a problem, but you need to truly understand what the problem you're trying to solve is for it to be fully effective.

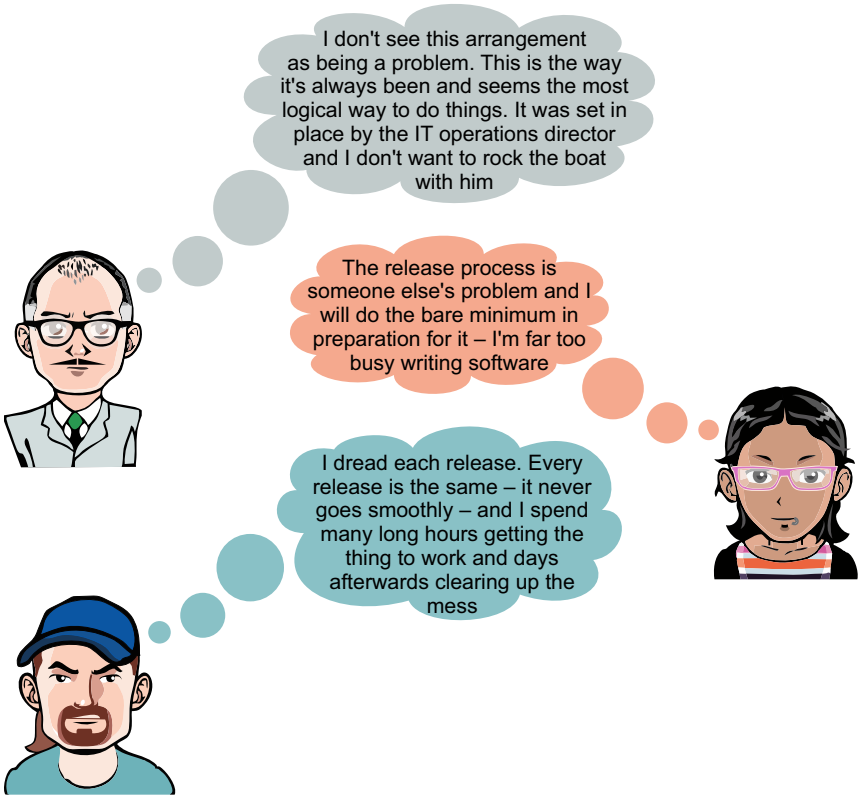
ACME systems took the time to understand the problem(s) they had before they began to implement CD and DevOps. They had to inspect before they could adapt.

Your first reaction to this might be that you don't have any problems, that everything is working well, and everyone involved with your software delivery process is highly effective, engaged, and motivated. If this is indeed true, then either:

- You have achieved software delivery utopia
- You are in denial
- You may not fully understand how efficient and streamlined software delivery can actually be

It's more likely that you have a workable process to deliver software, but there are certain teams of individuals within the process that slow things down. This is, most probably, not intentional; there might be certain rules and regulations that need to be adhered to, certain quality gates that are needed, it might be that no one has ever questioned why certain things have to be done in a certain way and everyone carries on regardless, or it might be that no one has highlighted how important releasing software actually is.

Something else to take into account is the fact that different people within your organization will see (or not see) a problem in different ways. Let's go back to ACME for a moment and examine the views of the three personas you were introduced to in relation to having the software releases controlled by the operations team:



As you can see, different people have wildly different views depending on what part they play in the overall process.

For the sake of argument, let's assume that you do indeed have some problems releasing your software with ease and want to understand what the root cause is (or most likely, what the root causes are) so that you can make the overall process more efficient, effective, and streamlined. Just like ACME, before you can *adapt*, you need to *inspect*; this is the fundamental premise of most agile methodologies.

Throughout this chapter, we will explore:

- How to identify potential issues and problems within your software delivery process
- How to surface them without resorting to blame
- How it can sometimes be tough to be honest and open, but doing so provides the best results
- How different people within your organization will see the same problem(s) in different ways

Before we start looking into how to *inspect*, I would like to go off on a slight tangent and talk about a large gray mammal.

Elephant in the room

Some of us have a very real and worrying ailment that blights our working lives: elephant in the room blindness, or to give its medical name, *Pachyderm in situ vision impairedness*. We are aware of a big problem or issue that is in our way, impeding our progress and efficiency, but we choose to either accept it, or worse still, ignore it. We then find ingenious ways to work around it and convince ourselves that this is a good thing. In fact, we might even invest quite a lot of effort, time, and money in building solutions to work around it.

To stretch this metaphor a little more—please bear with me, there is a point to this—I would like to turn to the world of art. The artist Banksy exhibited a piece of living artwork as part of his 2006 *Barely Legal* exhibition in Los Angeles. This living artwork was in the form of an adult Indian elephant standing in a makeshift living room with floral print on the walls. The elephant was also painted head to toe with the same floral pattern. The piece was entitled, as luck would have it, *Elephant in the Room*. It seems ludicrous at first, and you can clearly see that there is a massive 12,000 lb. elephant standing there; while it has been painted to blend in with its surroundings, it is still there in plain sight. This brings me to my point; the problems and issues within a software delivery process are just like the elephant, and it is just as ludicrous that we simply ignore their existence.

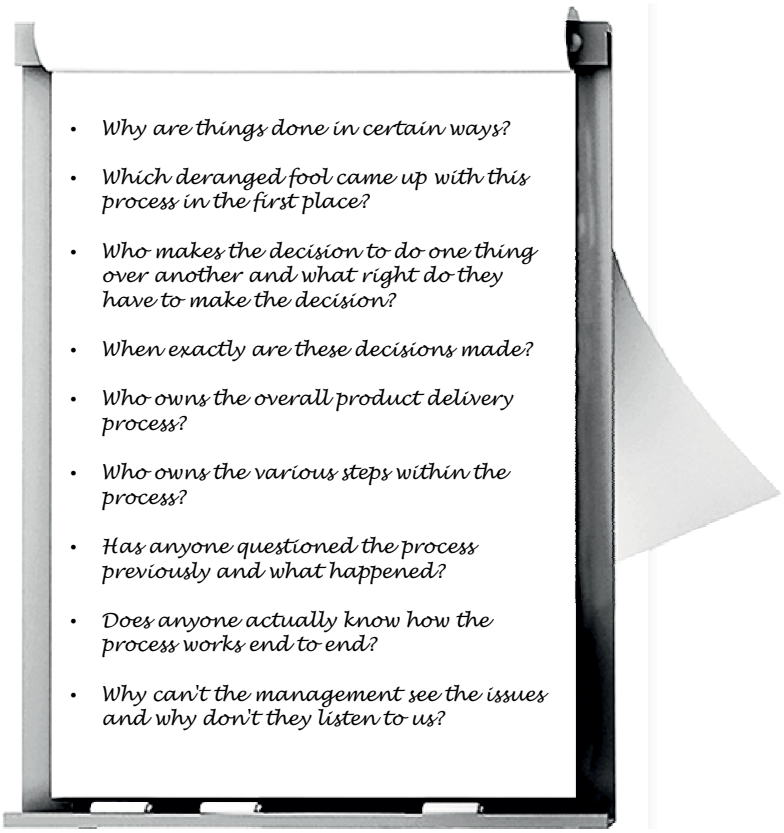
The elephant in the room is not hard to spot if you look closely. It's normally sitting/lurking where everyone can see. You just need to know how to look and what to look for before you can expose it.

Through the remainder of this chapter, we'll go through some ways to help highlight the existence of the elephant in the room and, more importantly, how to ensure as many people as possible can also see it and realize that it's not something to be avoided, worked around, or ignored.

Before you start removing the figurative floral pattern from the figurative elephant, there's still some legwork you need to do.

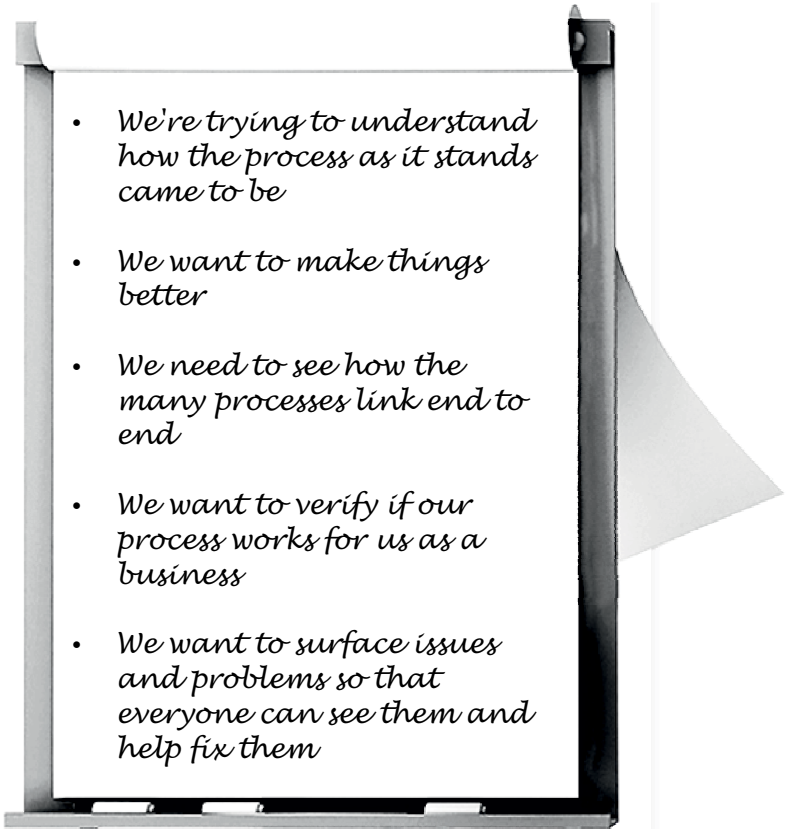
Defining the rules

With any examination, exposé, investigation, or inspection, there will be, to some degree, dirt that will need to be dug up. This is inevitable and should not be taken lightly. The sort of questions that will be asked may well include the following:



These types of questions may well make some people very uncomfortable and bring to light facts that produce emotive responses or emotional reactions, especially from those that might have originally had a hand in designing and/or implementing the very process that you are putting under scrutiny. Even if they can see and understand that the process they nurtured is broken, they might still have an emotional attachment to it, especially if they have been involved for a long time. You need to be mindful that these self-same people might be needed to help replace and/or refine the process, so tread carefully.

To keep things on a purely professional level, you should map out some ground rules that clearly define what the investigation is for and what its goal is. These rules need to be clear, concise, easy for everyone involved to understand, and worded in a positive way. The sort of things you should be looking at are as follows:



To further ensure you minimize the emotional reactions, you should define some rules of engagement so that everyone involved understands where the *line* is and when it is about to be crossed. Again, keeping these rules simple, concise, and using positive language will help everyone understand and remember them. Good examples would be:

Retrospection can be a powerful tool, and if used incorrectly, it can cause more trouble than good; you can shoot yourself in the foot many, many times. You need to make sure you know what you are letting yourself in for before you embark on this sort of activity.

You now need to consider who should be involved and who will add the most value.

Including (almost) everyone

Although you will have the best intentions and will want to include everyone involved in the software delivery process to take part in the inspection, this might be neither realistic nor practical. What you need is information from individuals who can actively contribute, are engaged, are ideally open to change (or at least would like to see things change for the better), and understand and agree to the aforementioned rules.

These engaged contributors should come from all parts of the business; if they are involved in product creation and delivery, they should be involved. You need a broad set of information and data to move forward; therefore, you need to get a broad set of people involved.

As you start compiling the list of participants, which for a large organization can be quite daunting, you will no doubt find that there will be some degree of natural selection as you start to ask people to contribute; some might say they're too busy, some won't want to be involved for reasons they don't want to disclose, and some might simply not care enough either way.

Identifying the key people

One tip when compiling the list of participants is to try and identify the *key people* within the process. These key people might not be obvious at first, however, asking simple questions such as *who should I ensure I invite to this?* or *who do you normally talk to if there's a problem?* of a number of different people from different parts of the business will give you some indications.

There is a strong certainty that some of these key people will be the ones who say they are too busy. The fact that they are too busy might be directly attributed to the fact that the process they are working within is broken, but they don't have the time to stop and realize this. I recommend that you take time to ensure that the key people who fit into this category are encouraged, cajoled, and convinced to take part; if they are key, it sometimes helps to let them know this, as an ego boost can sometimes help win them over. Again, playing the *if you don't take this opportunity to sort things out for the better, someone else might and it might be worse* card sometimes works.

You might (will) also come across individuals who are very eager to be involved simply because they have an axe to grind or need a soapbox to proclaim their personal opinion. This is fine, but you need to be mindful of the fact that such people can potentially derail the investigation process—which again might be why they want to be involved. You should not simply dismiss these people out of hand as they might have valuable observations to bring forward, and dismissing them might foster further negativity. You should, however, ensure these individuals agree to be engaged contributors and understand the ground rules you have set. Of course, you will need to keep an eye on them, much like the naughty children of the class. That said, you might be surprised at how much value they bring to the process.

Too many cooks

As you build your list of participants, you might encounter a positive problem—you have too many people who want to be involved in the investigation. In some respects, this is a good thing; oversubscription is a nice problem to have. If this is the case, you should consider running multiple sessions rather than drop people off the list. We'll cover the format of the session(s) in more detail later, but suffice to say they can turn out to be very interactive with a high degree of active participation. As such, my advice would be to try and keep the numbers down, otherwise you will end up with too many voices generating too much noise. What you want is a few voices providing valuable information and data. The rule of thumb would be a maximum of 30 participants, which is more than enough, unless you're a very experienced facilitator.

In essence, you need to engage and include as many people as possible who are actively involved in the process of defining, building, testing, shipping, and supporting software within your organization. The wider the net is cast, the more relevant the information and data you will catch, so to speak.

Not only is the way in which the investigation is to be conducted and who is involved very important, but it is also vitally important that you ensure the environment is set up correctly and the proper open and honest behaviors are used and encouraged throughout.

We'll look into behaviors and culture in more detail in a later chapter, but for now, let's concentrate on three key areas.

Openness, transparency, and honesty

To truly get to the bottom of a given problem, you need to gather as much information and data about it as possible so that you can collectively analyze and agree the best way to overcome or address it—you need to know how big the elephant truly is before you expose it and then remove it. The natural reaction of most management "Stans" will be to run a closed session investigation and invite a selected few to take part.

This might provide some information and data, but it's a strong bet that it will not give you what you need; things will be missed, people will feel intimidated, and they will not feel free to disclose some pertinent pieces of information; some might simply forget an important detail, or worse still, some of the information might be misinterpreted or simply taken out of context.

All in all, closed session investigations are a hotbed for distrust, nondisclosure, disengagement, and blinkered points of view. Therefore, it is not recommended that you follow this course.

To realistically get the level of information and engagement, you need to create an open and transparent environment in which positive behaviors and honest, constructive dialog can flourish. This does not mean you have to work within a glass house, have every conversation in public, and every decision made by a committee. What is needed is *a distinct lack of secrets*.

Let me clarify what I mean by using a couple of examples:

1. Your senior management begrudgingly admits that there might be a few problems that need addressing. They then instruct the VP of engineering to handpick a team of people they trust to compile a list of solutions to present back. The VP is under orders to not disclose or discuss this with anyone outside of the closed group.
2. Your CEO invites every employee to an all-day workshop and asks everybody to provide open and honest feedback about the issues they face day to day. The CEO and the leadership team then spend the next few weeks openly working through all of the feedback. A follow-up workshop is then arranged to honestly discuss and debate the various options available.

I think it's plain to see the difference, which of the two approaches will bear fruit, and which will wither and die.

All of this might sound unrealistically simple, but without openness, honesty, and transparency, people will remain guarded, and you will not get all of the facts you need—the word *facts* in used intentionally. You need an environment where anyone and everyone feels that they can speak their minds, and more importantly, contribute.

Location, location, location

Ideally, you should plan to run your investigation collocated (everyone in the same physical location) as this allows for greater human interaction, building rapport, building trust, and encourages the general ebb and flow of conversation in what can be highly interactive exercises. You might also want to consider running these sessions on neutral ground (for example, rent a conference room in a local hotel or office complex), which not only puts people at ease but also takes some focus away from the office and its day-to-day distractions.

If you don't have the luxury of collocated teams, you need to be a little more creative in how you approach things. You should consider the following:

- Bringing the remote team(s) to you, budget permitting
- Sending the local team(s) to them, again budget permitting
- Using video conferencing (voice conferencing just isn't good enough)

You should also ensure that you take into account challenges around time zones and come up with workable options. For example, don't expect your Boston-based team to remotely attend a workshop at 5 a.m. (EST) just because it's easier for the UK team.

As you can see, before you embark on the challenge of exposing the elephant in the room, there is some preparation you need to do.

Throughout this chapter, you have been introduced to terms such as *investigation*, *elephant exposure*, and *retrospection*. In relation to your software delivery process, these all mean pretty much the same thing: gathering information and data on how the process works end to end so that you can highlight the issues, which can then be rectified. We'll now move on to some of the ways you can gather this information and data, but before we do so, let's clear a few things up.

It's all happy-clappy management waffle – isn't it?

Some of you of a technical ilk might be reading this chapter wondering who it's targeted at and thinking *surely this is all soft skill people management self-help waffle and doesn't really apply to me*. In some respects, this is very true; any good manager or leader worth their salt should at least know how to approach this type of activity. However, you have to take into account the very real fact—an ineffectual process has a greater impact on those within it than those who are perceived to be running it. Simply put, as an engineer, if your effectiveness, morale, and enjoyment of your role is impacted by a process that you feel is broken and needs changing, then you have as much right and responsibility to help change it for the better as anyone else. In my experience, the best engineers are those who can examine, understand, and solve complex problems—be they technical or not. In addition, who better to have on board while trying to find out the problems with a process than those directly involved in it and affected by it?

If you're a Devina or an Oscar stuck in a process that slows you down and impacts your ability to effectively do your job, then I strongly encourage you to get actively involved with investigating and highlighting the problems (there will be many, and some might not be as obvious to you as you first think). It can be daunting, and yes, if you're employed to analyze requirements or design systems, cut code, or look after the infrastructure, then why should you get involved in a *business analysis*? It's simple really; if you don't do anything, then someone else might, and things might get worse.

If you're a Stan, then I suggest you actively allow and encourage all of your Devinas and Oscars to get involved. As we just stated, they are the ones who are living within the process, and by implication, they know the process very intimately—far better than you, I suspect. Yes, some might need your help, some might need encouragement, some might need training or coaching, some might need empowerment, and some might need all of the above. In the long run, it will be worth it.

Not only should you encourage the *troops* to be actively involved, you should also use your influence and encourage your peer group to do the same. On the face of it, this might seem easy to achieve, but it can be quite a challenge, especially when other managers start putting roadblocks in your way. The sort of challenges you will face include:

- Convincing them it is a good and worthwhile thing to do
- Getting them to agree to allow many people to stop doing their day jobs for a few hours so that they can participate
- Getting them to agree to listen to others and not drive the agenda
- Getting them to be open and honest within a wider peer group
- Ensuring that they allow subordinates to be open and honest without the fear of retribution
- Getting them to trust you

As you can imagine, you might well be involved in many different kinds of delicate conversations with many people, roles, and egos across the business. It won't be easy, but the perseverance will be worth it in the long run.

Now that we have cleared that up, let's move on to the fun part—exposing the elephant.

The great elephant disclosure

Let's presume at this point that you have overcome all of the challenges of getting people in the same location (physically and virtually), you have obtained buy-in from the various management teams, have agreed some downtime, and have a safe environment set up in a neutral venue. You're almost ready to embark on the elephant disclosure, almost. What you now need to do is actually pull everyone together and run the workshop(s) to capture the data you need. These types of sessions need two things:

- The staple toolset of any agile practitioner: some big blank walls covered in paper, large whiteboards, flipcharts, sticky notes, various colored pens, and various colored stickers, some space, plenty of biscuits, and a little bit of patience
- A tried-and-tested agile technique that defines the format for the workshop

With regards to the second point, there are many varied-and-proven techniques and exercises you can use with wonderful names such as *StoStaKee*, the Starfish, the Sailboat, and TimeLine.

For the sake of space, I'll include references to these (and many others) within *Appendix C*, *Retrospective Games*, and we'll focus on the one in particular that has proven to be effective.

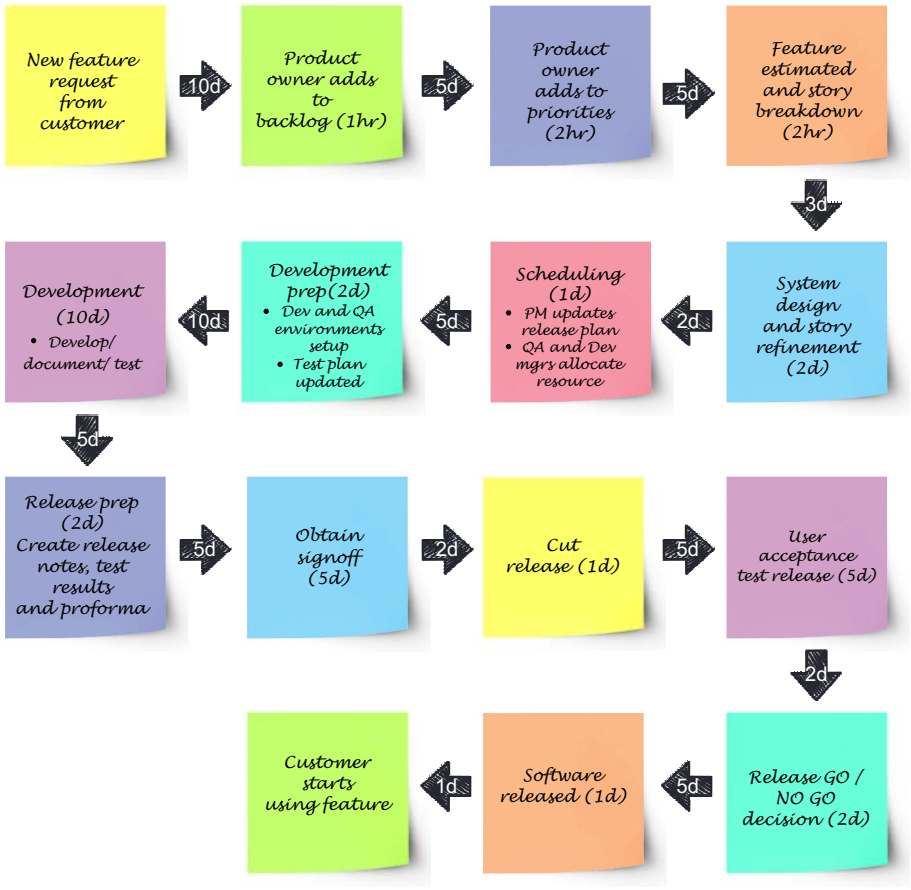
Value stream mapping

This lean technique derives from—as quite a few agile methodologies and tools do—manufacturing, and it has been around, in one guise or another, for many years. As with any lean methodology/tool/technique, value stream mapping revolves around a *value* versus *waste* model. In essence, a **value stream map** is a way to break down a product delivery process into a series of steps and handover points; it can also be used to help calculate efficiency rates, if that's useful to you. The overall map can be laid out and analyzed to see where bottlenecks or delays occur within the flow; in other words, you can see which steps within the process are not adding value. The key metric used within value stream mapping is the lead time (for example, how long before the original idea starts making hard cash for the business).

There are lots of resources and reference materials available to detail how to pull together a value stream map, and there are a good number of specialists in this area should you need some assistance.

To effectively create a value stream map, you will need a number of people across all areas of the business who have a very clear and, preferably hands-on, understanding of each stage of the product delivery process—sometimes referred to as the product life cycle. Ideally, a value stream map should represent a business process; however, this might be too daunting and convoluted at first. To keep things simple, it might be more beneficial to pick a recent example project and/or release and break this down.

As an example, let's go through the flow of a single feature request delivered by the ACME system's Version 2.0 business (before they saw the light):

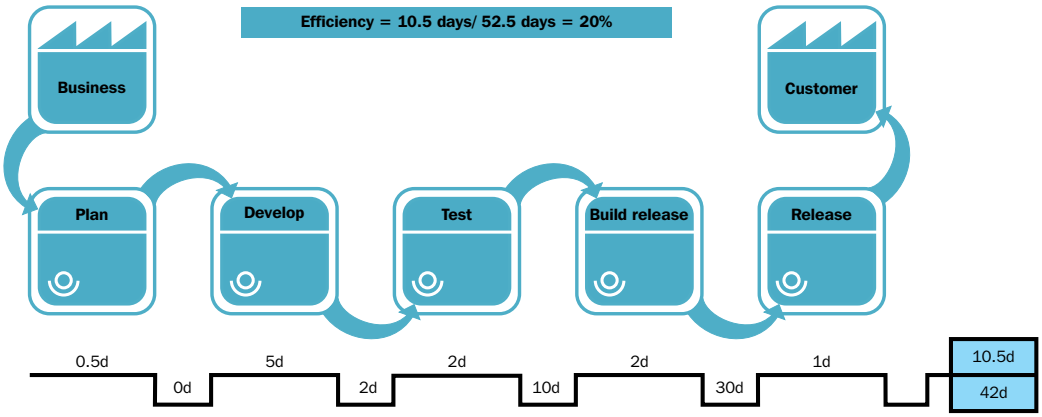


Each box represents a step within the overall process. The duration value within each box represents the working time (that is, how long it takes to go through each step). The duration value in each arrow represents the wait time between each step (that is, how long it takes between each step).

This is a very simplistic overview, but it does highlight how much time it can take to get even the most simplistic requirement out of the door. It also highlights how much waste there is in the process. Every step, delay, and mistake has a cost. The only real value you get is when the customer actually uses the software so if it takes too long to get a feature, then the customer may well get bored of waiting and go elsewhere.

On the face of it, generating this type of map should be quite simple, but it can also be quite a challenge. This simplistic diagram is created in real time with input from many different areas of the business. There will be lots of open and honest dialogue and debate as facts are verified, memories jogged, dates and times corroborated, examples clarified, and agreements reached across the board as to what actually happens.

If you prefer to use the standard value stream mapping terminology and iconography, you can take the sticky notes version and convert it into something like the following, which again represents the flow of feature requests through the business:



This diagram also includes the efficiency (which is based upon the amount of time value is being added versus the dead time within the flow)

The most valuable output from this particular technique is that you can stop the obvious areas of waste. These are the parts of the overall process that are slowing down and impeding your overall ability to deliver. With this information, you can focus on these problem areas and start to look at options that will make them less wasteful and more valuable to the overall process.

Summary

Throughout this chapter, you have been given an insight into the following aspects: how to expose problems within your product delivery process—what we are calling the elephant in the room, the challenges and benefits of using collaborative, engaging approaches to identify these problems, and some effective tools and techniques to help you break down the problems into easily identifiable chunks of work.

Now, you know how to obtain valuable information and data about your problem(s) and have some much-needed actions to work with. You now know how to inspect. Let's presume these problems revolve around the waste created through long release cycles and a siloed organization. This being the case, you have a very clear objective, which will almost certainly address the problems and deliver what the entire business needs—you need to implement a CD and DevOps ways of working. All you now need to do is pull together a plan to implement it. In other words, you now need to adapt; this is handy as that's what we'll cover in *Chapter 3*, *Plan of Attack*.

3
Plan of Attack

Throughout *Chapter 2*, *No Pain, No Gain*, which for ease of reading we'll now refer to as the *inspect* stage, you were introduced to the tools and techniques to identify the problems you may well have with your overall product delivery process. We referred to this as the "elephant in the room" as it is something that is not hard to spot, just very easy to ignore. The presumption here is that the problems identified are the common place issues related to most software delivery processes. Some of these issues are listed as follows:

- Waste due to having too many handover and decision points in the process
- Waste due to unnecessary wait time between steps
- Many software changes are packaged up into large, complex *big bang* releases
- Large and infrequent releases breed an environment for escaped defects and bugs
- Releases are seen as something to dread rather than a positive opportunity for change
- People are disengaged or there is low morale (or both)
- Software changes are not trusted until they have been tested many many times
- Over complex dependencies within the software design
- Tasks that are duplicated throughout the process

We will now take the information and data you have captured and work on the method of turning this into something that you can implement to overcome the problems—a plan of attack to implement CD and DevOps if you will.

This plan of attack should not be taken lightly; just like the *inspect* stage, there is quite a bit of groundwork you need to do to ensure the scope of the implementation is understood, accepted, and communicated. As with any plan or project, there needs to be an end goal and a vision of how to get there.

Setting and communicating the goal and vision

A goal and vision for any project is important as it ensures all concerned know what is expected, and for those working on the project understand where it and they are heading. It may sound simple but it is not always obvious. How you communicate the goal and vision is just as important as setting them. Do this incorrectly and you are in danger of losing buy-in from the business, especially the senior management. For example, they may believe that to fix just one or two of the issues highlighted during the *inspect* stage will be enough to overcome all of the problems found. You have to be crystal clear what you plan to achieve, and crystal clear who you are communicating this to.

When it comes to CD and DevOps, this can be quite challenging as the deliverables and benefits are not always easy for the uninitiated to understand or envision. It may also be difficult to fully quantify as some of the benefits you obtain from the adoption of CD and DevOps are not wholly tangible (that is, it is quite hard to measure increases in team collaboration and happiness).

The best advice is **keep it simple stupid** (**KISS**). You have a list of issues that the wider business has provided, and what they want is something (anything) that will make their lives easier and allow them to do their jobs. If truth be told, you most probably have more things on the list than you can effectively deliver. This should be seen as a good thing as you have some wriggle room when it comes to prioritization of the work.

Your challenge is to use this list and pull together a goal and vision, which will resonate with all of the stakeholders and ensure it is something that can be delivered. This may need quite a bit of effort but it is doable. For a good example, let's once again have a look at ACME systems. When they were planning the implementation of CD and DevOps, they came up with a goal for the project, which was *to be able to release working code to production 10 times per day*. This was a nice simple tag line, which everyone could understand (almost everyone, but we'll come to that later) and formed the basis of their communication strategy. Yes, it was an ambitious goal but they knew with a little hard work, courage, determination, and the right people involved, it was possible. They even created posters that could be stuck on walls around the office.

A very simple tag line that anyone and everyone can understand

Setting your goal may be just as easy. You have a good understanding of the business problems that need to be overcome, you know which teams are involved, and you have a good idea of what will resonate with the stakeholders. The goal needn't be a grandiose thing—maybe something as simple as to allow engineers *to release their own code* or *ship code at the press of a button* will suffice. The most important thing here is to set a goal that people can get behind, which solves some—ideally most—of the problems highlighted.

Once you have a goal (or most probably, a list of potential goals), canvass opinion from people whose judgment you trust; if they think your proposal is way off the mark, it might just be so. If you're lucky enough to have PR or marketing people accessible, canvass their opinions; this is after all something they are pretty good at.

Once you're happy with the goal, you need to work on the vision. It may help to think of the goal as *what* you want to achieve and the vision as *how* you will achieve it. The vision should contain as much detail as can be easily communicated and understood. You have to be mindful of the fact that too much detail will confuse and cause people to become disengaged, so KISS.

Let's go back to the ACME team to see how they went about doing this. They had a goal (*release working code to production 10 times per day*) and now had to set out the vision. They took the list of issues highlighted during the inspect stage and translated them into a list of easy-to-understand actions and tasks that needed addressing. A good example of this is the problem *engineers are unable to ship their own code*, which made perfect sense to the engineers themselves, but wasn't simple enough to understand for anyone outside of that group—questions such as *ship to where?* and *what do you mean by ship?* arose. After some debate, discussion, and refinement, this problem was translated to *allow engineers to release fully working code from their laptop to the live platform with ease*. This was something the majority could understand—it was simple. This can also be simplified further to *laptop to live*, which still conveyed the meaning but was easier to digest and communicate.

The vision ACME created included a wide variety of things, some technical and some not, which could all be clearly communicated and understood. Those of you who are au fait with agile ways of working, may spot this as *a prioritized feature backlog*.

The next step was to share the goal and vision to the business and stakeholders and gain an agreement that what was proposed would address the problems and issues captured during the inspect stage. This was directed to as wide an audience as possible—not just the management—with many sessions booked over many days to allow as many people to be involved as possible.

Once the goal and vision had been shared, discussed, and revised—based upon the constructive feedback from all involved, a top-level plan was pulled together. Put simply, ACME generated a story backlog that contained almost everything they needed to address and deliver.

To ensure transparency and ease of access to the goal and vision, the ACME team needed to ensure that all of the data, information, and plans were publicly (internal to the business rather than in the public domain) available. To this end, they fully utilized all of the internal communication and project repository tools available to them: internal wikis, blogs, websites, intranets, and forums.

If you don't have tools like these available to you, it shouldn't be a vast amount of effort to get one set up using open source solutions. There are even on-line solutions that are secure enough to keep company secrets safe—some examples can be found in *Appendix A, Some Useful Information*. Having this level of transparency and openness will help as you move forward with the execution of the plan. This is especially true of *social* solutions such as blogs and forums, where feedback can be given and virtual discussions can take place.

It all sounds pretty simple when it's put down into a few paragraphs and to be honest, it could be with the right environment and the right people involved. It's just a case of ensuring that you have a good grasp of what the business and stakeholders want, you know how to summarize this into an easily understandable goal and align the vision to drive things in the right direction. The key here is *easily understood*, which can sometimes be a challenge, especially when you take into account communication across many business areas (and possibly many time-zones and cultures), where each have their own take on the terminology and vocabulary used. This brings us to how you should communicate and ensure everyone involved understands what is happening.

Standardizing vocabulary and language

One small and wholly avoidable factor that can scupper any project is the misinterpretation of what the deliverables are. This may sound a little alarming but projects can fail simply because one person expects something, but another person misunderstands or misinterprets and delivers something else. It's not normally down to ignorance; it's normally due to both sides interpreting the same thing in different ways.

For example, let's look at something simple—the word *release*. To a project manager or a release manager, this could represent a bundle of software changes, which need to be tested and made live within a schedule or program of work. To a developer working in an agile way, a release could be a one line code change, which could go live soon after he/she has completed coding and running the tests.

There can also be a bit of a problem when you start to examine all of the different words, terminology, and **three letter acronyms** (**TLA**) that we all use within IT. We therefore need to be mindful of the target audience we are communicating to and with. Again the KISS (a FLA or four-letter acronym if you prefer) method works well here. You don't necessarily have to go down to the lowest common denominator; that may be very hard to do and could make matters worse. Try to strike a balance. If some people don't understand, then get someone who does understand to talk with them and explain; this will help bridge the gap and also helps to form good working relationships.

Another suggestion to help bridge the gap is to pull together a glossary of terms that everyone can refer to. The following is a simple example:

Term	**What it is**	**What it is not**
Continuous Delivery	A method of delivering fully working and tested software in small incremental chunks to the production platform	A method of delivering huge chunks of code every few weeks or months
DevOps	A way of working that encourages the Development and Operations teams to work together in a highly collaborative way towards the same goal	A way to get developers to take on operational tasks and vice versa
CD	See Continuous Delivery	
Continuous Integration	A method of finding software issues as early as possible within the development cycle and ensuring all parts of the overall platform talk to each other correctly	Something to be ignored or bypassed because it takes effort
CI	See Continuous Integration	
Definition of done	A change to the platform (software, hardware, infrastructure, and so on) is live and being used by customers	Something that has been notionally signed off as something that should work when it goes live
DOD	See definition of done	
Release	A single code drop to a given environment (testing, staging, production, and so on)	A huge bundle of changes that are handed over to someone else to sort out
Deploy	The act of pushing a release into a given environment	Something the Operations team does

If you have a wiki/intranet/blog/forum, then that would be a good place to share this as others can update it over time as more buzzwords and TLAs are introduced.

The rule of thumb here is to ensure whatever vocabulary, language, or terminology you standardize on, you must stick to it and be consistent. For example, if you choose to use the term *CD and DevOps*, you should stick with it through all forms of communication, written and verbal. It then becomes ingrained and others will use it day to day, which means conversations will be consistent and there is much less risk of misinterpretation and confusion.

You now have a goal, a vision, a high level backlog, a standard way of communicating, and you're ready to roll (almost). The execution of the plan is not something to be taken lightly. Whether you are a small software shop or a large corporate, you should treat the adoption and implementation of CD and DevOps with as much gravitas as you would any other project, which touches and impacts many parts of the business. For example, you wouldn't implement a completely new e-mail system into the business as if it were a small scale *skunk works* project—it takes collaboration and coordination across many people. The same goes for CD and DevOps.

A business change project in its own right

Classing the implementation of CD and DevOps as business change project may seem a bit dry but that's exactly what it is; you are potentially changing the way the whole business operates, for the better. Not something to be taken lightly at all. If you have ever been involved in business change projects, you will understand how far reaching they can be.

There's a high probability that the wider business may not understand this as well as you do. They have been involved in the investigation and have verified the findings and seen what you intend to do to address the issues raised. What they may not understand fully is the implication of implementing CD and DevOps—in terms of the business, it can be a life-changing event. A little later on in the book, we'll go through some of the hurdles you will face during the implementation. However, if you have a heads-up from the start, you're in a much better position to leap over the hurdles.

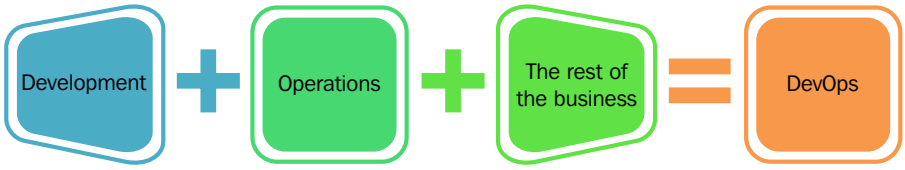
Suffice to say that you should ensure you get the business to recognize that the project will be something that will impact quite a few people, albeit in a positive way. Processes will change as will the ways of working. We're not just talking about the software delivery process here; CD and DevOps will change the way the business thinks, plans, and operates.

For example, let's assume that marketing and program management teams are currently working on a 6 to 9 month cycle to take features to the market. If all goes well with the CD and DevOps implementation, they will have to realize that a feature may be live in a few weeks or even days. They will therefore need to work at a different cadence and will have to speed up and streamline their processes. From experience, this sort of change also brings with it some unexpected benefits—that being a renewed level of trust throughout the business that when the development and Operations team say they will deliver something, they actually deliver it. Therefore, the traditional contingency plan B is no longer required (nor plans C, D, or E). The way features are delivered will drastically change and the rest of the business needs to accept this and be ready for it.

In the early stages of the project, the wider business will most probably believe that the impact of CD and DevOps—as the name suggests—will be localized to the Development and Operations teams. The following diagram depicts the extent of this change as the business sees it:

What the business sees at the early stages

At first, this may not be too far from the truth, and you may start small so that you can get to grips with the subtleties and to find your feet as it were. This is fine; however, once you get some momentum—which won't take long—things will start to change very quickly and if people aren't ready, or at least aware, you may hit some barriers, which could slow things down or even stop the implementation in its tracks. The business therefore must accept that the impact will be far reaching as depicted below:



What the business should be seeing as representative of the areas that will be impacted and involved

Getting the business to understand this will not be an easy task, they will need some convincing and some good old-fashioned diplomacy may be again required. Luckily, CD and DevOps is now becoming more widely known outside of the traditional IT realm and there is plenty of information, such as case studies, available to reference. That said, you have to be mindful that the wide business will still see this as an *IT thing* rather than a *business thing*.

Let's move forward and presume that the business is in agreement regarding the wide reaching nature of the implementation and (almost) everyone is behind the project. The next challenge is looking at the merits of using a dedicated team to implement the goal and vision.

The merits of a dedicated team

As with any high-profile project, it's worth considering the merits of having a dedicated team focused on the implementation of CD and DevOps. This is the approach ACME took and although it worked for them, it's your call whether you go down this route or not. There may be a temptation to run the implementation as a *skunk works* project. This type of project will tend to bubble along in the background and be staffed by like-minded individuals who have an interest but don't have the backing or reach needed to make sweeping changes, nor the free time to dedicate to make it truly successful. My recommendation is to not do this as such projects tend to fade away as more seemingly important projects—which do have backing and formal widespread recognition—take the limelight and more importantly the resource. This may sound cynical but it's a fact of life and you need to be mindful of this.

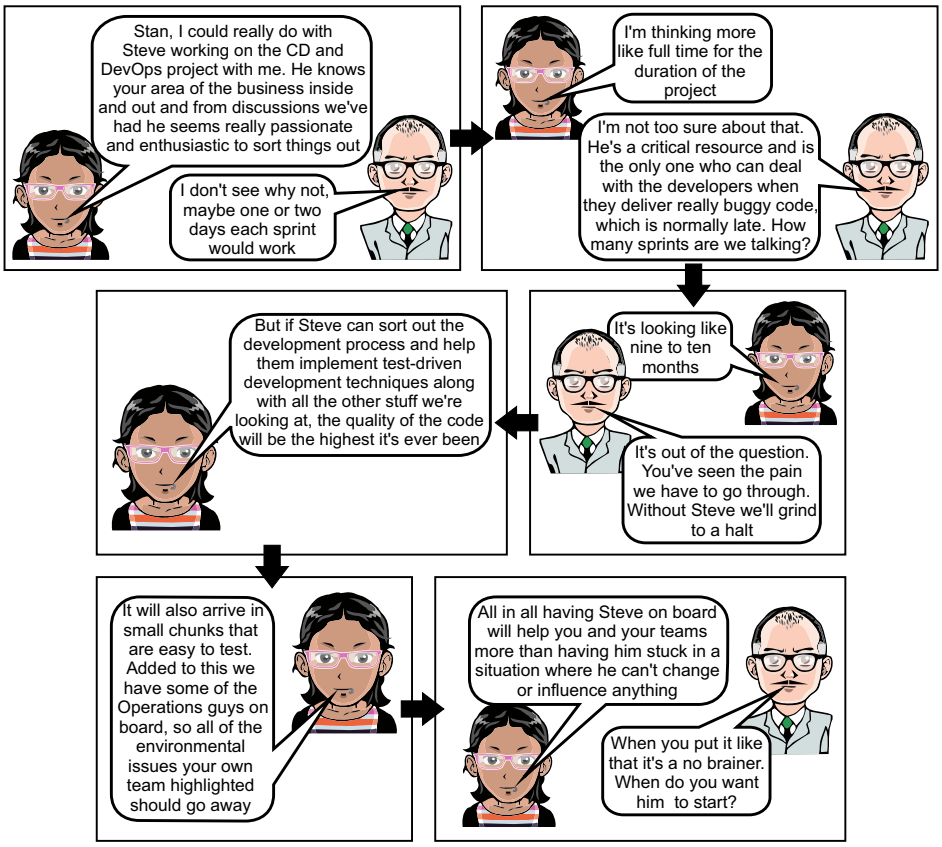
There seems to be a growing trend to hire or set up dedicated CD and/or DevOps teams to run the process of delivering software. This is not what I am referring to. I am referring to setting up a cross-functional and multi-disciplined team that can help drive the goal and vision on behalf of the wider business.

I would advise that you work with the wider business to ensure their agreement to the implementation of CD and DevOps goes further than simply paying lip service. They need to put their money where their mouth is and provide some people to help with the implementation. Reiterating the fact that the implementation of CD and DevOps should be considered as a business change project may help. In the end, you will need a team of like-minded individuals working on the project from across the business (not just developers and operations guys) to make this successful. You may want to start small and build up but the reality is that once things start getting traction, the wider business will need to get involved.

As soon as this is highlighted, you will no doubt get some areas of the business take a big step back in terms of engagement and commitment—especially those areas that manage the very people you want to second onto the project. It is then down to you to cajole, beg, bargain, and barter to get the people you need. To be honest, this shouldn't be too difficult as you have quite a large amount of ammunition to use—the same information and data you worked so hard to compile, which the business itself agreed was causing pain. If you used the value stream mapping exercise, you should also be able to pinpoint the pain areas with accuracy.

One thing to take into account is the level of commitment of those outside of the traditional IT realm—I'm thinking of such things as sales, marketing, HR, finance, and so on. It may not be viable to have such individuals dedicated full-time to the project. However, you should ensure they allocate some of their time and make themselves available when needed.

Let's take a typical discussion between you - who, for this example, is played by Devina and Stan, the manager—who, for this example, is the head of Testing and QA:



I admit it might not go exactly along those lines but you can see the point. You have been given a clear insight into what pains the business and have been asked to remove the said pains. The business needs to realize that this will not come without some cost and that they need to provide you with what you need to get the job done.

Who to include

If you decide to utilize a dedicated team, then you'll no doubt want to know who to include. This really depends on the way in which your business is set up. To help a little, let's again go back to ACME systems and see what they did. During version 2.0 of their evolution, the teams that were actively involved in delivering software included: Architecture, Development, Testers, Operations, and Change Management. They therefore decided to include individuals from each of these disciplines within their CD and DevOps team. They then added a scrum master and a product owner and topped it all off with a senior manager (someone who could act as the project sponsor and represent the team at a higher level). To follow the theme of inclusion, they also added stakeholders from the wider business. In the end, the ACME systems CD and DevOps team looked something like this:

The ACME CD and DevOps project team

If and how you go about setting up your own CD and DevOps team and who you include totally depends on the way in which your business is set up. The most important thing to remember when/if setting up a dedicated team is that it must be made up of more than just developers if they are to have credibility and be successful.

Let's move on from the practical elements of team building to the weird and wonderful world of evangelism and the benefits this activity can bring.

The importance of evangelism

Whenever you introduce a change, be it a new product, service, or a wholesale change to the ways of working, you need evangelists to ensure everyone who needs to know about the change, knows about it. Sometimes, this is seen as marketing or PR, but in its basic form, it is evangelism. Evangelism is important. It's also very hard work but is essential for any change to be successful. CD and DevOps has the potential to introduce a vast amount of change to the way your business works. Therefore, a vast amount of evangelism will be required. Even if you have a goal, vision, and the blessing from up on high, you need to evangelize to ensure those who are most important to the success of the implementation are behind you and understand what they are getting behind. You need to get out there and be seen and be heard.

Don't get me wrong, to evangelize across an entire business is going to take some effort and some determination. It will also take some energy. Actually, that's wrong; it will take a lot of energy. Your target audience is wide and far reaching, from senior management to the shop floor. So, it will take up quite an amount of time for you to get the message across. Before we go into the details of what to say to who, when, and how, let's get the ground rules sorted:

- If you are to be convincing when evangelizing to others the virtues of CD and DevOps, you need to believe in it 100 percent—if you don't, then how can you expect others to?
- You and whoever is involved in the project must practise what you preach and lead by example. For instance, if you build some tools as part of the project, make sure you build and deploy them using the exact same techniques and tools you are evangelizing about.
- Many people will not get it at first. So, you will have to be very, very patient. You might have to explain the same thing to the same person more than once. Use these kind of individuals as a yard stick; if they start to understand what CD and DevOps is all about, then there's a pretty good chance your message is correct.

- Remember your target audience and tailor your message accordingly. Developers will want to hear technical stuff, which is new and shiny; system operators would want to hear words such as *stability* and *predictability*; and management types would want to hear about *efficiencies*, *optimized processes*, and *risk reduction*. This is rather generalist. However, the rule of thumb is if you see their eyes glaze over, your message is not hitting home, then change it.
- Some people will simply not want to know or listen and it may not be worth focusing your efforts to make them (we'll be covering some of this in *Chapter 6*, *Hurdles Along the Way*). If you can win them round, then kudos to you but don't feel dejected by a few laggards.
- Keep it relevant and consistent. You have a standardized language, a goal, and a vision so use them.
- Don't simply make stuff up. Just stick to what can be delivered as part of your goal and vision; nothing more, nothing less. If there are new ideas and suggestions, get them added to the backlog for prioritization.
- Don't on any account give up.

What it boils down to is you will need to talk the talk and walk the walk. There will be quite a bit of networking going on; so be prepared for lots of discussion. As your network grows, so will your opportunities to evangelize. Do not shy away from these opportunities, and make sure you are using them to build good working relationships across the business as you're going to need these later on. Evangelizing is rewarding and if you really believe that CD and DevOps is the best thing since sliced bread, you will find that having opportunities to simply talk about it with others is like a busman's holiday.

As mentioned earlier, evangelism is a form of PR. So, if you have PR people available (or better still as part of the team), you should also look into getting simple things together, such as a logo or some *freebies* (such as badges, mugs, mouse mats, and so on), to hand out. This may seem a little superfluous but as with any PR you will want to ensure you get the message across and have it imbedded into the environment and peoples' psyche.

Up until this point, I may have painted things in a somewhat rosy glow. Adopting CD is no picnic. There's quite a big hill to climb for all concerned. As long as everyone involved is aware of this and has the courage and determination to succeed, things should go well.

Courage and determination

Courage and determination may seem like strong words to use but they are the correct words. There will be many challenges, some you are aware of some you are not, that will try to impede the progress. So, determination is required to ensure this keeps moving in the right direction. Courage is needed as some of these challenges will require you, the team, and the wider business to make difficult decisions, which could result in actions being taken from which there is no going back. I'll refer to ACME systems Version 2.0 for a good example of this.

In the early days of their adoption of CD and DevOps, they started with a small subset of their platform as the candidates for releasing using the new deployment toolset and ways of working. Unfortunately, at the same time, there was a lot of noise being generated around the business as another release (using the old *package everything up and push out as one huge deployment* method) was not going well. The business asked everyone to focus on getting the large release out at all costs, including halting the CD trials. This didn't go down too well with the team. However, after a rather courageous discussion between the head of the ACME CD team and his peers, it was agreed that resource could be allocated if there was universal agreement that this would be the last of the *big bang* releases and that all future releases would use the new CD pipeline process going forward. The agreement was forthcoming and so ended the era of the big bang release and the new era of CD dawned. After the last of the *big bang* releases was eventually completed, the entire development and operations teams were determined to get CD up and running as soon as possible. They had been through enough pain and needed another way or rather a better way. They persevered for a few months until the first release, using the new tooling and ways of working, went to the production environment, then the next, and so on. At this point, there was no turning back as too much had changed.

As you can no doubt appreciate, it took courage from all parts of the business to make this decision. There was no plan B and if it hadn't worked, they had no way to release their software. Knowing this fact, the business was determined to get the new CD and DevOps ways of working imbedded and established.

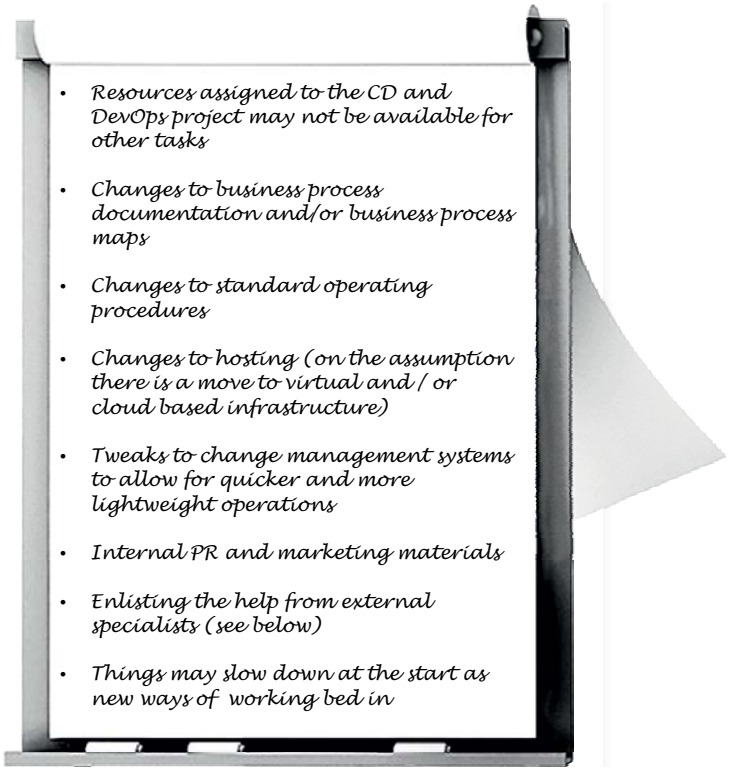
The preceding example could be classed as an extreme case but nonetheless, it goes to show that courage and determination are sometimes very much needed. If there's a will, there's a way.

Before we move away from the planning stage, there are still a couple of things you should be aware of as you prepare to embark on your new adventure: where to seek help and ensuring you and the wider business are aware of the costs involved with implementing CD and DevOps. We'll cover costs first.

Understanding the cost

Implementing CD and DevOps will ultimately save the business quite a lot of money—that is a very simple and obvious fact. The effort required to release software will be dramatically reduced, the resources required will be miniscule when compared to large *big bang* releases, the time to market will be vastly reduced, the quality will be vastly increased, and the cost of doing business (that is, volume of bug fixes required, support for system downtime, fines for not meeting SLAs, and so on) will be negligible. That said, implementing CD and DevOps does not come for free. There are costs involved and the business needs to be aware of this.

Let's break these down:



These costs should not be extortionate; however, they are costs that need to be taken into account and planned for. As with any project—especially one as far reaching as CD and DevOps—there will always be certain costs. If the business is aware of this from the outset, then the chance of it scuppering the project later down the line can be minimized.

There may be some costs that are indirectly caused by the project. You may have some people who cannot accept the changes and simply decide to move on; there will be costs to replace them (or not as the case may be). As previously stated at the beginning of the transition from *big bang* releases, you may well slow down to get quicker. If you have contractual deadlines to meet during this period, it may be prudent to renegotiate them.

You will know your business better than anyone—especially, after completing the investigations into how the business functions—so, you may have better ideas related to costs. Just make sure you do not ignore them.

Let's now focus on where you can get help and advice should you need it.

Seeking advice from others

Before you take the plunge and change the entire way your business operates, it would be a good idea to do some research and/or reach out to others who:

- Have been through this transition already—maybe a few times
- Have insights that you may not considered or thought about
- Have some battle scars that you will want to avoid
- Are in the same boat as you

There is an ever-growing number of people around the globe who have experience in implementing (and even defining) CD and DevOps. Some are experts in the field and focus on this as their full-time jobs; some are simply members of the growing community who have seen the light and selflessly want to help others realize the benefits they have witnessed and experienced.

If you can secure the budget to have an external expert come in to work with you that may help take some of the pressure off. If you do go down this route, be mindful of who you chose—remember that some *consultancies* will be more than happy to assist and will be just as happy to sell you their latest CD pipeline tool. If you're confident in your approach, then maybe just having a sounding board available every few weeks/months will suffice.

To reiterate, implementing CD and DevOps is no picnic and sometimes being at the forefront can be a lonely place. Do not feel like you should struggle alone. There are some valuable reference materials available (this book being one of those I would hope) and more importantly, there are a good number of communities—online and face-to-face meet-ups—which you can join to help you.

You never know, your story and input may be an inspiration for others, so in true DevOps style, break down the barriers and enjoy open and honest dialogue. I'll include a list of some of the reference materials and contacts in *Appendix A*, *Some Useful Information*.

Summary

As with any business change project, to successfully implement CD and DevOps, you need to ensure you know *what* you are setting out to do (your goal), understand *how* you're going to reach it (your vision), understand who's help you will need, clarify how you will communicate and evangelize, be realistic about how much it will cost, and most important of all, be realistic about how much work, courage, energy, and determination it will take. As mentioned previously, this is no picnic but it will be worth it.

Hopefully, you're all fired up and ready to go but before we do this let's take a look at something that may not be so obvious now but is yet another essential part of the CD and DevOps jigsaw: culture, and behaviors.

4
Culture and Behaviors

In *Chapter 2*, *No Pain, No Gain*, we learned that asking people to be open and honest is not easy unless you set the environment up to allow for it to happen. The culture and environment have to be such that honest disclosure can take place, and you have to ensure that every participant agrees to behave according to the flexible rules set out. We will now take this newfound knowledge and expand upon it to ensure the culture and behaviors throughout the organization are set up to allow for—what can be—potentially massive change. The sorts of things we'll cover throughout this chapter are:

- Why culture is so important
- How culture and behaviors affect your progress
- Encouraging innovation at grass roots
- Fostering a sense of accountability across all areas of your organization
- Removing blame from the working environment
- Embracing and learning from failure
- Building trust
- Rewarding success in the right way
- Instilling the sense that change is good and not risky
- How good PR can help

Throughout this chapter, we'll also look at what this means to the three personas (Stan the manager, Devina the developer, and Oscar the ops guy) you were introduced to in *Chapter 1*, *Evolution of a Software House*.

It should be noted that I am by no means an expert in the human sciences, nor do I have a PhD in psychology. What follows are learnings through observation, experience, and collaboration with experts in these fields.

Let's start by clarifying why culture is so important to CD and DevOps.

All roads lead to culture

Some people might think that CD and/or DevOps is simply about implementing technical tools, making slight tweaks to existing heavyweight processes to allow software to be released every few weeks, or worse still, a bona fide reason to set up a new "DevOps" team inside an existing organization. If you think any of these are correct, then you are wrong. CD and DevOps are—put very simply—ways of working. As such, the culture in which people work and the behaviors they exhibit has a massive part to play. If you have barriers or power struggles between teams, silos across the organization, ineffective lines of communication, a rigid old-school hierarchy, dysfunctional leadership, or your business is resistant to change or learning from failure, then your environment and culture are not conducive to adopting CD or DevOps ways of working. Attempting to implement CD or DevOps in such an environment will ultimately lead to failure, unless you address the underlying behaviors and overarching culture.

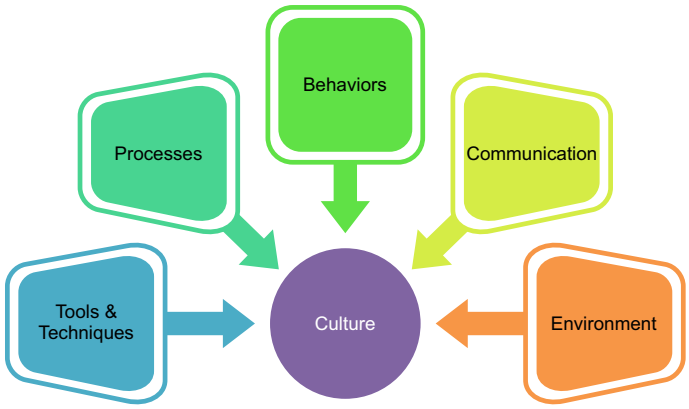
As we found in *Chapter 2*, *No Pain, No Gain*, which for ease of reading we'll now refer to as the *inspect* stage, culture is quite nebulous and can be hard to define. The following diagram attempts to clarify what we mean by culture in relation to this subject; we'll cover some of this in more detail throughout this chapter.

The cultural interconnectedness of all things CD and DevOps

You need to ensure that your organization is set up in such a way as to allow for all of the positive behaviors you witnessed during the *inspect* stage to resurface and become the norm. In essence, what you need is a positive, collaborative, and open culture. This is no mean feat, but it can be done. To some degree, you have already planted the seed during the *inspect* stage—albeit in safe greenhouse conditions—and have proven and realized the benefits. What you need to do is nurture this seedling and let its roots dig deep and spread across your organization.

This Cultural Revolution shouldn't be restricted to the shop floor; whoever is involved in or makes decisions about software delivery, be they an engineer, a manager, or the VP of engineering, needs to at least have an understanding and appreciation of how culture and behaviors can help or hinder the adoption of CD and DevOps.

A healthy culture is central to a successful way of working, and is therefore central to CD and DevOps, as is depicted in the following diagram:



Culture is central to all aspects of successful CD and DevOps adoption

Now, let's revisit something that we looked at during the setting up of the *inspect* stage and expand upon it.

An open, honest, and safe environment

Apart from sounding like something taken directly out of a management training manual, what does having an open, honest, and safe environment actually mean? In relation to CD and DevOps, this means that you need to ensure that anyone and everyone involved in your product delivery process is able to openly comment and discuss ideas, issues, concerns, and problems, without the fear of ridicule or retribution.

As you found during the *inspect* stage, allowing for open discussions and honest appraisals of how things are done within your organization and the product delivery process brings to the surface details and facts that otherwise would have been missed or stayed hidden. You need to persist the culture where the distinct lack of secret behavior is maintained or, if there is a gap between your *inspect* stage and implementation stage, rekindled and reaffirmed.

On the face of it, this all sounds like common sense, but unfortunately, this way of working is not encouraged, or worse still, is actively discouraged in some working environments. If you find yourself in this situation, then you have some additional challenges to overcome simply due to the fact that these edicts are normally defined and enforced through the HR and management guidelines, which in-turn define the policies under which the business operates. You therefore can't simply break or bend these rules at will. We'll cover this in more detail later in the book, but suffice to say that you need to tread very carefully and ensure you lead by example in terms of your behaviors.

Let's break down these concepts in more detail.

Openness and honesty

Openness and honesty are key factors to ensure that the implementation of CD and DevOps is successful. Without these behaviors in place, it's going to be very difficult to break down barriers and implement the much needed changes throughout your organization. You already engaged the majority of the business during the *inspect* stage to obtain honest feedback about the current situation. You now need to ensure that you continue this dialogue with all concerned. Everyone involved in the product delivery process, from developers and testers through change and release controllers to product owners and senior managers, must have a forum they can use to share their thoughts, suggestions, observations, worries, and news.

The most effective way to do this, as was the case during the *inspect* stage, is via face-to-face human interaction, be this in person or virtually via video conference systems (remember that video is preferable to voice). There is one potential drawback to this approach—getting everyone in the same place at the same time. We'll look at some ways to overcome physical environment challenges later; if face-to-face is not wholly viable all the time, you can look at other options such as online collaboration tools (Campfire, for example), real-time forums (Yammer, for example), or group chat systems (IRC or HipChat, for example). Links to the aforementioned tools can be found in *Appendix A*, *Some Useful Information*.

Whatever approach you choose, it is advisable that you set up some form of etiquette or guidelines so that everyone knows what is acceptable and what is not. Hopefully, common sense will prevail. What should not prevail is a heavy-handed policing or moderation of the content as this will discourage openness and honesty and ultimately make the solution redundant.

Let's look at what this means in terms of contribution from our three personas:

- Stan can use his influence to ensure his peers understand the importance of openness and honesty and that they encourage their teams to exhibit these behaviors.
- Devina and Oscar can work together to implement the aforementioned tools that can help enhance communications across the organization. They can also influence their peer groups to be more open and honest.

As you go through the implementation of CD and DevOps, it is extremely important that you have regular open, honest, and truthful feedback from all concerned in terms of what is working with the implementation and, more importantly, what is not. Again, the simplest and most effective way is face-to-face human interaction; simply walk around and ask people. Again, if this is not wholly viable, then you should consider sort of lightweight survey solutions (such as survey monkey). The word 'lightweight' is important here as no one will provide feedback on a regular basis if they have a 10-page questionnaire to fill out every few weeks.

If you follow or use an agile methodology and run regular retrospectives, ask those running these sessions to forward on any feedback related to your implementation.

You're hopefully getting an idea of what open and honest dialogue is all about, but what about courageous dialogue, where does this come into the equation?

Courageous dialogue

There will be times when someone lower down the food chain will have an opinion or a view on how those above them help or hinder the product delivery process. You might have individuals whose views are at odds with specific parts of the business, or indeed, other individuals. It takes guts for an individual to speak up about something like this, especially within a corporate environment. For these people to speak up, they want to be sure that what they say (within reason, of course) is not taken as a black mark on their record. They need to be given a **de-militarized zone** (**DMZ**), where they can share their ideas, views, and opinions—where they can point out the emperor's new clothes.

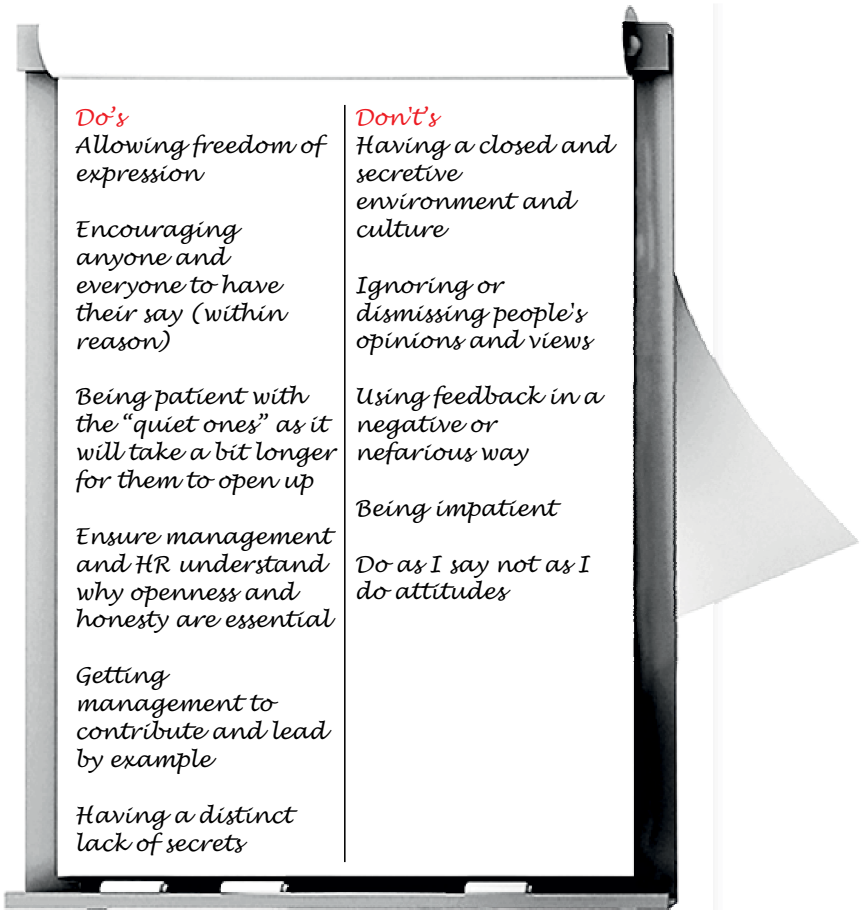
You should work with the management team, and HR, if need be, to ensure that there is a forum for this type of dialogue as it is very important. The content might not be enlightening, but if you have a number of people saying the same thing, then there is a good chance that something needs to be addressed. At the very least, you can work with the management types to implement some sort of amnesty or a way for anonymous feedback to be collected—a suggestion box or online surveys, for example.

One important thing to also take into account is the *quiet ones*. Generally speaking, there are two distinct types of personality traits: *introverted* and *extroverted*. The *extroverts* are the ones that are not afraid to interact, talk, and discuss their views and feelings in public. For *extroverts*, open, honest, and courageous dialogue isn't something they would shy away from. *Introverts* are the opposite, and will more often than not simply close down or just go with the flow in these situations. You, therefore, need to be very mindful of this fact and ensure everyone has the opportunity to contribute and voice their opinions. It might seem like additional work, but from experience, it will be worth it as the contributions from the *introverts* are normally well considered and enlightening.

If you have difficulty spotting the different types, then here's one easy tip: *extroverts* talk to make their brains work whereas *introverts* use their brains to make their mouths work.

Let's be very open, honest, and courageous about how easy it will be to implement and embed these sorts of behaviors into normal ways of working—it is not. It will be challenging, complex, time consuming, and at times, very frustrating. However, if you persevere, and it starts to work (and it will), you'll find it's a very effective way to work. You will find that once openness and honesty are embedded into the normal ways of working, things really start coming together.

Let's summarize what we've covered so far:



What might not be obvious is the fact that the physical environment is something that can and does cause further complications when looking at encouraging open and honest dialogue and behaviors. We'll now take a look at this.

The physical environment

Some of you might be lucky enough to work in nice, airy, open-plan offices with plenty of opportunity to wander around for a chat and line-of-sight visibility of people you collaborate with. The reality is that you might not be so lucky and will have teams physically separated by office walls, flights of stairs, the dreaded cubicles of doom, or even time zones. At this point, let's hypothesize that the office space is not open-plan and there are some physical barriers. There are a few things you can look at to remove some of these barriers:

- Keep the office doors open or, if possible, remove them altogether.
- Set aside an area for communal gatherings (normally in the vicinity of the coffee machine) where people can chat and chill out.
- Have regular (weekly) sessions where everyone gathers (normally in the vicinity of coffee and doughnuts) to chat and chill out.
- Get a foosball table; it's amazing how much ice is broken by having a friendly foosball competition within the office.
- If you use scrum methodology (or similar) and have separate teams locked away in offices, each holding their daily stand-up in private, then hold a daily scrum of scrums (or stand-up of stand-ups), and have one person from each team attend it.
- See whether some of the partition walls can be removed.
- If you have cubicles, remove them, all of them. I personally think that they are the work of the devil and produce more of a negative environment than having physical walls separating teams.
- See whether an office move-around is possible to get people working closer together, or at the very least, mix things up.
- As previously mentioned, look into implementing some sort of collaborative forum/messaging/chat solution, which everyone can use and have access to. You can also inject a bit of fun and innovation using tools such as Hubot (`https://hubot.github.com`), which might encourage more people to use the solution.
- Stop relying on e-mail for communications and encourage people to talk—have discussions, mutually agree, and follow up with an e-mail, if need be.

These are, of course, merely suggestions based upon a very broad assumption of your environment, and you will no doubt have better ideas. The end game here is to start removing the barriers, both virtual and physical.

Let's see what our personas can do to help:

- Stan, the manager, can work within his peer group to convince those above of the importance of changes to the physical environment. Trying this alone, especially when money needs to be spent, might be challenging, so having many *management* voices saying the same thing will add weight.
- Devina and Oscar can work together to make small changes and run experiments, for example, to be seen to have face-to-face discussions, rather than via e-mail, or take over an area of the office and sit together.

We'll now move on from the seemingly simple subject of openness and honesty to the seemingly simple area of collaboration.

Encouraging and embracing collaboration

As you set out on your journey to implement CD and DevOps, you will no doubt be working with the assumption that everyone wants to play ball and collaborate. Most of the business has been through an exercise to capture and highlight the shortcomings of the incumbent process, and you all did this together in a very collaborative way; surely they want to continue in this vein?

At first, this might well be very true; however, as time goes on, people will start to fall back into their natural siloed position. This is especially true if there is a lull in the CD and DevOps implementation activity—you might be busy building tools or focusing on certain areas of the existing process that are most painful. Either way, old habits will sneak back in if you're not careful.

It is, therefore, important that you keep collaborative ways of working at the forefront of people's minds and encourage everyone to work in these ways as the default mode of operation. The challenge is to make this an easy decision (for example, working collaboratively is easier to do than not).

You must keep your eyes and ears open to ensure you get an early indication when things slip back. If you have built up a network, use it to find out what's happening.

There are many ways to encourage collaboration, but you need to keep it lightweight and ensure that those you are encouraging don't feel that this way of working is being forced on them; they need to feel it's their idea. Here are some simple examples:

- Encourage everyone to use your online collaborative forum/messaging/chat solution rather than e-mail—even incentivize its use at first to get some buy-in.
- If the norm is for problems to be discussed at a weekly departmental meeting, rather than having a 5-minute discussion at someone's desk, cancel the departmental meeting and encourage people to get up and walk and talk.
- If the norm is *headphones on and heads down*, you should discourage this as it simply encourages isolation and stifles good old-fashioned talking to each other. If people like to listen to music while working, you can consider something radical, such as a jukebox or some Sonos/networked speakers.
- Even if you don't follow a scrum methodology, use the daily stand-up technique across the board—you can even mix it up across teams so people can move between the stand-ups.
- Install some magnetic whiteboards around the office space, which will encourage people to get up, mix, and be creative while explaining problems, showing progress, or simply having fun and doodling.
- Ensure you mingle and keep open discussions with all teams—you never know, you might hear something that another person has also been discussing, and you can act as the CD and DevOps matchmaker.

Besides impacting openness and honesty, the physical environment can impact (positively and negatively) the adoption of collaborative ways of working, so you need to be mindful of this.

Let's again see what our personas can do to help:

- Collaboration is not the exclusive realm of engineers. Managers can and should be seen to collaborate. Stan should be actively seen to be collaborating, and if tools have been implemented, he and his peers should actively use them.
- Devina and Oscar should practice what they preach and be highly visible when collaborating. Even simple things such as encouraging developers and operations engineers to go to the same pub on a Friday lunchtime can make a difference.

As collaboration becomes embedded within the business, you will see many changes come to life. At first, these will be quite subtle, but if you look closely you'll soon start to see them: more general conversations at peoples' desks, more *I'm trying to do X but not sure of the best way – anyone fancy a chat over coffee to look at the options?* in the online chat room, and more background noise as people talk more or share the joke of the day.

Some subtle (or sometimes not so subtle) PR might help, for example, posters around the office, coffee mugs, or even prizes for the most collaborative team; anything to keep collaboration in sight and mind.

Let's leave collaboration for now and together move on to innovation and accountability.

Fostering innovation and accountability at grass roots

If you're lucky enough to work (or have worked) within a technology-based business, you should be used to having innovation as an important and valued input for your product backlog and overall roadmap. Innovation is something that can be very powerful when it comes to implementing CD and DevOps, especially when this innovation comes from the grass roots.

Many of the world's most successful and most used products have come from innovation, so you should help build a culture throughout the business where innovation is recognized as a good and worthwhile thing rather than a risky way of advancing a product. Most engineers thrive, or at least enjoy, innovation, and if truth be told, this was most probably one of the major drives for them choosing to become engineers—this and the fine wine, fast cars, and the international jet-setter lifestyle (okay, this might be stretching things a bit too far).

This isn't to say that they can all go off and do what they want; there are still products to deliver and support. What you need to do is allow some room for investigation and experimentation—rekindle the R in R&D. Innovation is not just in the realm of software; there might be different ways of working or product delivery methodologies that come to light that you can and should be considering.

Agile techniques such as Test-driven development (TDD), scrum, and Kanban all started out as innovative ideas before gaining wider notoriety.

Despite normal convention, innovation is not the exclusive right of solutions and systems architects; anyone and everyone should be given the opportunity to innovate and contribute new ideas and concepts. There are many ways to encourage this kind of activity (competitions, workshops, and so on), but you need to keep it simple so that you get a good coverage across the business. One simple idea is to have a regular innovation forum or get-together, which allows anyone and everyone to put forward, and if possible, prototype an idea or concept.

Innovation can increase risk, new things always do; therefore, the engineering teams must understand that with the freedom they are given to make decisions and choices comes responsibility, ownership, and accountability for the *new stuff* they come up with, produce, and/or implement. They cannot simply implement shiny new toys, tools, processes, and software and hand them off to someone else to support. The *somebody else's problem* (*SEP*) or *throw it over the wall* approaches will no longer work.

A good example of this is the ACME systems plan to allow developers to deploy code directly to production. On the face of it, this is very much what CD and DevOps is all about, but one simple question caused the plan to falter. The question was *who is going to hold the pager?*, or to bring this into the 21st century, *are the developers going to be on call when things go wrong out of hours?* Ultimately, you need everyone involved in the process of delivering and supporting software to have the same strong sense of accountability so that the question need not be asked.

So, how can these values and behaviors be instilled into your organization? Let's see what our personas can do to help:

- Stan should actively allow time for his team members to *try things out* or *experiment*, be this by setting aside some 10 percent time or simply encouraging them to put forward their ideas and suggestions for product or productivity advancement.
- Devina and Oscar should actively pursue this agenda as part of discussions with their managers during one-to-one's or team meetings. To help things along, using some spare time on an idea, and then presenting it back, might be a good thing as it shows commitment and that you're serious. Working together collaboratively will also add credence.

As your adoption of CD and DevOps matures, you will find that innovation and accountability will become commonplace as the engineering teams (both software and operations) will have more capacity to focus on new ways of doing things and improving the solutions they provide to the business. This isn't just related to new and shiny things; you'll find that there is renewed energy to revisit the technical debt of old to refine and advance the overall platform.

Believe it or not, sometimes things will go wrong. We'll now look at how things that don't go so well should be dealt with, and why a culture of blame is not a good thing to have.

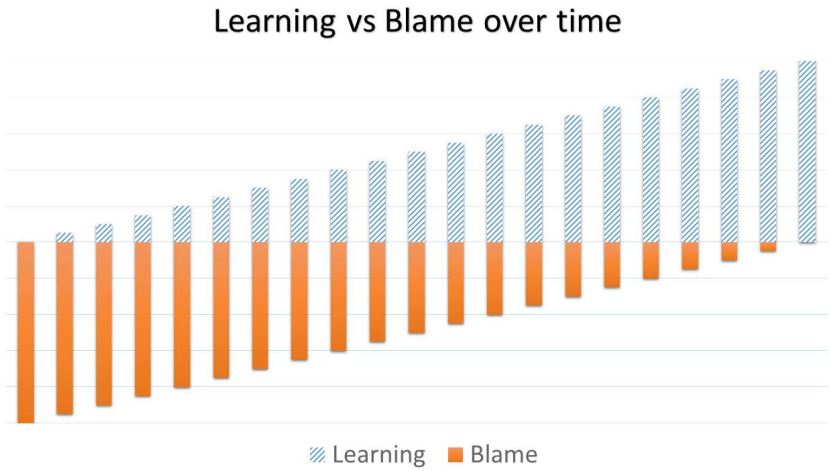
The blame culture

Encouraging a fail-fast way of working is a critical element to good agile engineering practice; it is all well and good saying this, but this has to become a real part of the way your business works; as they say, actions speak louder than words. If, for example, we have a manager who thinks that pointing the finger and singling people out when things go wrong is a good motivational technique, then it's going to be very difficult to create an environment where people are willing to put themselves out and try new things. A culture of blame can quickly erode all of the good work done to foster a culture of openness, honesty, collaboration, innovation, and accountability.

Ideally, you should have a working environment where when mistakes happen (we're only human and mistakes will happen), instead of the individual(s) being jumped on from on high, they are encouraged to learn from the mistake, take measures to make sure it doesn't happen again, and move on. No big song and dance. Not only this, but they should also be actively encouraged to share their experiences and findings with others, which enforces all the other positive ways of working we covered so far.

Blame slow, learn quickly

In a commercial business, it might sound strange and be seen as giving out the wrong message (for example, you might seem to be ignoring or encouraging failure), but if lessons are being learned, and mistakes are being addressed quickly out in the open, then a culture of diligence and quality will be encouraged. Blaming individuals for a problem that they quickly rectify is not conducive to a good way of working. Praising them for spotting and fixing the issue might seem wrong to some, but it does reinforce good behaviors. The following illustration shows the possible impact of a *blame slow, learn quickly* culture:

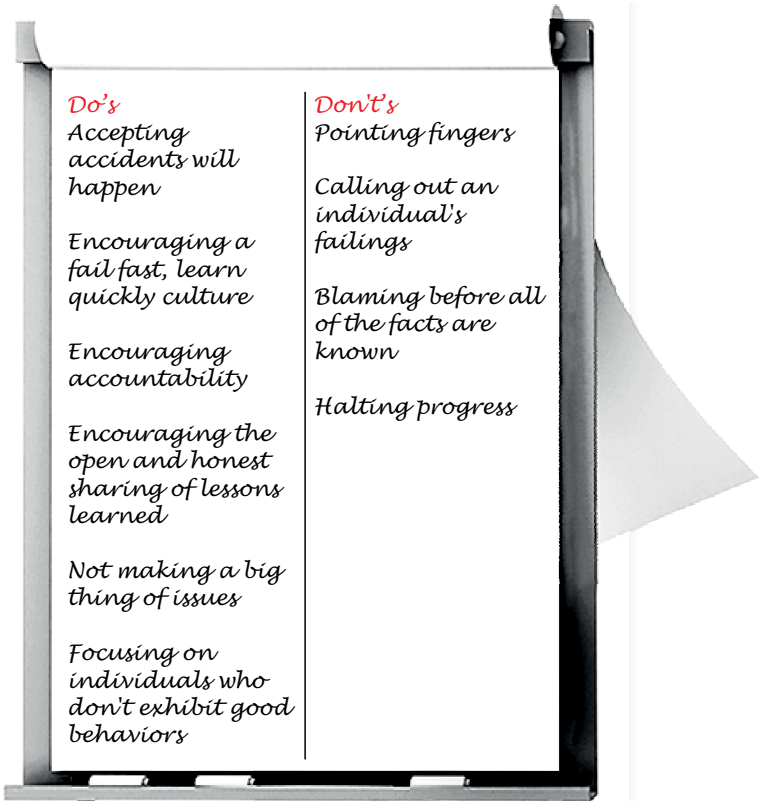


As blame diminishes, learning will grow as people will no longer feel that they have to keep looking over their shoulders and only stick to what they know or are told to do

If managers are no longer preoccupied with the small issues, they can focus on the individuals who create issues but don't fix them or take accountability.

As you can no doubt understand, this culture change is not going to be easy for some, especially for the managers who have built up the reputation of being Mr. or Mrs. *Shouty*. Sometimes, they will adapt, and other times, they might simply step out of the way of progress—as the groundswell gains momentum. They will have little choice but to do one or the other.

Let's again summarize this:



Removing the threat and culture of blame from the engineers' working life will mean that they are more engaged, willing to be more open and honest about mistakes, and more likely to want to fix the problem quickly.

Of course, there is a large element of trust required on all sides to make this work effectively.

Building trust-based relationships across organizational boundaries

Now, I will freely admit that this does sound like something that has been taken directly from an HR or management training manual; however, trust is something that is very powerful. We all understand what it is and how it can benefit us. We also understand how difficult things can be with a complete lack of it. If you have a personal relationship with someone, and you trust them, the relationship is likely to be open, honest, and a long and fruitful one. Building trust is extremely difficult; you don't simply trust a colleague because you have been told to do so—life doesn't work this way. Trust is earned over time through peoples' actions.

Trust within a working environment is also a very hard thing to build. There are many different reasons for this (insecurity, ambition, reputation, personalities, and so on), so you need to tread carefully. You also need to be patient as it's not going to happen overnight.

Building trust between traditional development and operations teams can be even harder. There is normally a level or an undercurrent of distrust between these two areas of the business:

- The developers don't trust that the operations team know how the platform actually works or how to effectively investigate issues when they occur.
- The operations team don't trust that the developers won't bring the entire platform down by implementing dodgy code.

This level of distrust can be deeply ingrained and is evident up and down the two sides of the business. These types of attitudes, behaviors, and the culture they create are all too negative. It's hard enough to get software developed, shipped, and stable without playing silly games with who does what and who doesn't. If you have an environment like this, then the business needs to grow up and act its age.

There is no silver bullet to forge a good trust-based relationship between two or more factions; however, the following techniques proved to work for ACME systems and might help you:

- If you arrange for some off-site CD or DevOps training, ensure that you get a mix of software and operations engineers to attend and ensure they are in the same hotel. You will be amazed how collaborative working relationships start out in the hotel bar.
- If there are workshops or conferences you are looking at attending (for example, DevOpsDays), make sure there's a mix of Devs and Ops in attendance and a hotel bar.

- If you are a manager, be very mindful of what promises and/or commitments you make and ensure you either deliver against them or you are very open and honest as to why you didn't/couldn't.
- If you are an engineer, act in exactly the same way.
- If you have set up an innovation forum (as mentioned previously), encourage all sides to attend and contribute.
- Discourage *us and them* discussions and behaviors.
- If it's viable, try and organize job swaps or secondments across the software and operational engineering teams (for example, get a software engineer to work in operations for a month, and vice versa). This can also include management roles.

We'll now move from trust to rewards and incentives.

Rewarding good behaviors and success

How many of us have worked with or been part of a business that throws a big post-release party to celebrate the fact that against all odds you managed to get the release out of the door? On the face of it, this is good business practice and management 101, and after all, most project managers are trained to include an *end of project party* task and budget in their project plans. This is not altogether a bad thing if everything that was asked for has been delivered on time to the highest quality. Let's try rewording the question.

How many of us have worked with or in a business that throws a big post-release party to celebrate the fact that against all odds you managed to deliver most of what was asked for and only took the live platform offline for 3 hours while they tried to sort out some bugs that had not been found in testing?

If the answer to the question is *quite a few, but it was a hard slog and we earned it*, then you are a fool to yourself. Rewarding this type of behavior is 100 percent the wrong thing to do. The businesses that deliver what customers want and do it quickly are the ones that succeed.

If you want to be a business that succeeds, you need to stop giving out the wrong message. We did say that it was okay to fail as long as you learn from it quickly; we didn't however mention rewarding failure to deliver. You should be rewarding everyone when they deliver what is needed when (or before) it is needed. The word *everyone* is quite important here as a reward should not be targeted at an individual as this can cause more trouble than it's worth. You want to instill a sense of collaboration and DevOps ways of working, so make the reward a group reward, a party, a day out, and so on.

The odd few

Okay, so there might be the odd few who will put in extra effort when times get sticky, and rewarding those individuals is not a bad thing; however, this should not be the norm. If engineering teams (software and operational) are consistently being told to work long days, long nights, and weekends, then there is something wrong with the priority of the work. If, however, they decide to apply some extra effort to overcome some long outstanding technical debt or implement some labor-saving tools to speed things up, then this is completely different, and you should be looking at specific rewards for these specific good behaviors.

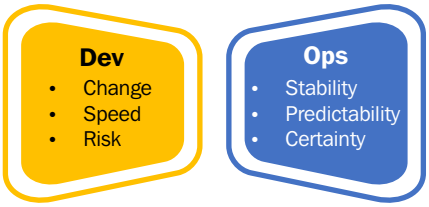
At the end of the day, you want to reward individuals or teams for doing something amazing that is above and beyond the call of duty, rather than simply successfully releasing software. As CD and DevOps ways of working become embedded, you will notice that you don't actually have what you would previously have called releases anymore (they are happening too quickly to notice each one), and therefore, you need to look at other ways to reward. For example, you can look at throwing a party when a business milestone is hit (such as when you reach the next millionth customer), when a new product successfully launches, or simply because it's sunny outside and the bosses want to say thank you.

CD and DevOps will change the way the business operates, and this fact needs to be recognized across all areas. As such, the way you reward people needs to change to instill the good behaviors previously mentioned (openness and honesty, innovation, accountability, and so on). This can be quite a shift for some businesses, and some might even need to implement new reward systems, solutions, or processes to cater for this.

One of the standard ways of rewarding people is via some kind of bonus or incentive scheme. This will also need to change, but first you need to recognize how the current system might foster the wrong behaviors and can stifle your implementation of CD and DevOps.

Recognizing dev and ops teams are incentivized can have an impact

There is a simple and obvious fact that some people might not instantly realize, but it is something that is very real and very widespread throughout the entire IT industry. This fact is that development teams are incentivized to deliver change, whereas operations teams are incentivized to ensure stability and system uptime, thus discouraging change. The following figure highlights this:



Incentivising developers to deliver more quickly is at odds with incentivising operations teams with keeping things stable and safe

If you think about it, the two methods of incentivizing are at odds with each other; operations teams get a bonus for reducing or even stopping change, and development teams get a bonus if they deliver a lot of change. So, how do you square the circle and allow a steady stream of change to flow through without having the operations team up in arms about the fact that their bonus is at risk?

There's no simple answer, but there are some examples you can look at to ease the pain:

Incentive	**Pros**	**Cons**
Have the same incentives across both Dev and Ops.	If you are incentivizing to allow for continuous change, you will increase the potential for having CD and DevOps becoming the norm as everyone involved will focus on the same goal.	There is more risk as people might think that changing things quickly is more important than quality and system uptime.
Including each side of the DevOps partnership in each other's incentive schemes.	If some of the bonus of the software engineering team is dependent on live platform stability, then they'll think twice before taking a risk. If some of the operations engineering team's bonus is dependent on enabling CD, they will think twice before blocking changes just for the sake of it.	If the percentage of the *swap* is small, it might be ignored as the focus will remain on getting the majority of the bonus, which will still encourage the old behaviors.
Replacing the current incentive scheme with one that focuses on good behaviors and encourages a DevOps culture.	This has the potential to remove conflict between the engineering teams (Dev and Ops) and would encourage them to focus on what is important: delivering products customers want and need.	The reality is that it will be quite difficult to get a full agreement, and get it in place quickly, especially in a corporate environment. This doesn't mean it's not something worth pursuing.

Whatever you do in regards to incentivizing and rewarding people, you need to instill a sense of positivity around change, while at the same time ensuring risk is reduced.

Embracing change and reducing risk

In the same vein as fostering innovation and accountability at grass roots, you need to work across the wider organization to ensure they accept the fact that change is a good thing and not something to be feared.

It is true to say that changing anything on the production platform—be it a new piece of technology, a bug fix to a 15-year old code-base, an upgrade to an operating system, or a replacement storage array—can increase the risk of the platform, or parts thereof, failing. The only way to truly eliminate this risk is to change nothing, or simply switch everything off and lock it away, which is neither practical nor realistic. What is needed is a way to manage, reduce, and accept the risk.

Implementing CD and DevOps will do just that. You have small incremental changes, transparency of what's going on, the team that built the change and the team that will support it working hand in hand, a sense of ownership and accountability from the individual(s) who actually wrote the code, and a focused willingness to succeed. The challenge is getting everyone in the business to understand and embrace this as the day-to-day way of working.

Changing people's perceptions with pudding

Getting the grass roots to understand this concept should be quite simple when compared to other parts of the business that are, by their very nature, risk averse. I'm thinking here of the QA teams, senior managers, project and program managers, and so on. There are a few ways to convince them that risks are being controlled, but the best way is via using the *proof of the pudding* methodology:

- Pick a small change and ensure that it is well publicized around the business
- Engage the wider business, focusing on the risk averse, and ensure they are aware; also invite them to observe and contribute (team stand-ups, planning meetings, and so on)
- Ensure that the engineers working on the change are also aware that there is a heightened sense of observation for the change
- As the change is progressing, get the engineering teams involved to post regular blog entries detailing what they are doing, including stats and figures (code coverage, test pass rate, and so on)

- As the release goes through the various environments to production, capture as many stats and measurements as possible and publish them
- When all is done, pull all the above into a blog post and a post-release report, then present them

You might be thinking that this is a vast amount of work, and to be honest, it is if you follow the preceding steps for each and every change you make. What it does do is serve a purpose; it proves to the business that change is good and risks can be controlled and managed. I recommend you follow these steps a few times to build trust and confidence—you can always refine later down the line. Another positive you will find is that it will foster a culture of diligence at grass roots; if they are very aware that the business is keeping an eye on things, especially when things go wrong, then they will think twice before doing something silly.

It should be noted that even though the steps detailed will generate additional work, this is nothing compared to how some organizations currently function; changes are fully documented and risk assessed, progress meetings are held, the project progress is publicized, and every meticulous detail is captured and documented. Is it any wonder why delivering software can be so painful?

As with anything in life, if you make a small change, the risk is vastly reduced. If you repeat the process many times, the risk is all but removed and habits are formed. To follow this thread, if infrequent releases contain a large amount of change, the risk is large. Make it small and frequent, and the risk goes away. It's quite simple when you look at it this way.

As part of the *proof of the pudding* example, there was a lot of publicizing and blog posting going on. This should not be seen as an overhead, but a necessary part of CD and DevOps adoption. Being highly visible is key to breaking down barriers and ensuring anyone and everyone is aware of what is going on.

Being transparent

As we previously covered, being secretive about what you do and how you do it is not conducive to building an open, honest, and trust-based working environment or culture. If anyone and everyone can see what is going on there should be no surprises. What we're looking for is a culture, and ways of working where change is good and frequent, individuals work together on common goals, the wider business trusts the product delivery teams to deliver what is needed when it is needed, and the operations teams know what is coming. If there is a high degree of visibility across the entire process, anyone and everyone can see this happening, and more importantly, how effective it is.

You should look at the option of installing large screens around the office to display stats, facts, and figures. You might well have something like this set up already, but I suspect these screens display very technical information—system stats, CPU graphs, alerts, and so on. I also suspect that most of these reside in the technical team areas (development, operations, and so on). This is not a bad thing, it's just very specialized, and those of a nontechnical nature might well ignore them or most likely not even know that they exist. See if you can move some of the screens to communal areas of the office or try and find some budget to buy new ones.

You should also complement this highly technical information with very simple, easy to read and understand data related to your CD and DevOps process. You should be looking at displaying the following kinds of information:

- Number of releases this day, week, month, and year against the number yesterday, last week, last month, and last year
- The release queue and progress of the current release going through the process and who initiated it
- Production system availability (current and historical)
- If you use an online scrum/Kanban board (such as **AgileZen** or **Trello**), then consider having this data displayed to show your backlog, work in progress, and work completed along with related stats such as velocity and burndown
- The latest business information such as share price, active user numbers, the number of outstanding customer care tickets, and so on

The last point is very important. You should publish, display, and advertise complementary information and data that is business-relevant, rather than simply focusing on technical facts and figures. This will help to heighten engagement and awareness outside of the technical teams. Having this information visible as you progress through your adoption and implementation of CD and DevOps will also provide proof that things are improving.

Summary

We covered quite a lot of ground in terms of the human side of implementing CD and DevOps throughout this chapter. Hopefully, it has been impressed upon you that the culture in which you operate is essential for CD and DevOps to work. When it comes to collaboration, you will find that trust, honesty, and openness are powerful tools that allow individuals to take responsibility and accountability for their actions. Rewarding good behaviors and removing blame will also help drive adoption.

At this point, you should have a plan and some insight into the importance of culture and behaviors when implementing CD and DevOps. In *Chapter 5*, *Approaches, Tools, and Techniques*, we'll look at some things that will help as you drive forward.

5
Approaches, Tools, and Techniques

The preceding chapters focused on surfacing the issues within your delivery process and defining a plan to address these by (hopefully) implementing CD and DevOps. The chapters also taught you how to set up the working environment to ensure that you're successful. The following chapters will focus on the steps you need to consider when executing against the *plan*. First, we will focus on some of the technical aspects.

There will be quite a lot of things to cover and take in, some of which you will need, some of which you might have in place, and some of which you might want to consider implementing later down the line. Whatever the situation, what follows should provide some small chunks of wisdom, or information at the very least, that you can adapt or adopt to better fit your requirements. I should point out that the majority of this chapter is focused on software engineering (that is, the Dev side of the DevOps partnership); however, quite a lot of the areas covered are as relevant to system operations as they are to software engineering.

In this chapter, we'll cover the following topics:

- Examples of proven engineering best practice
- Some simple rules and tools
- Automated CI and CD tooling
- The value of monitoring and metrics
- How a manual process can be just as effective as tools

It is worth pointing out at this point that the tools and approaches mentioned are not mutually exclusive—it is not a case of all or nothing—you just need to pick what works for you. If you want to look at the tools in more detail, take a look at *Appendix A*, *Some Useful Information*. Let's start with some good old-fashioned engineering best practice.

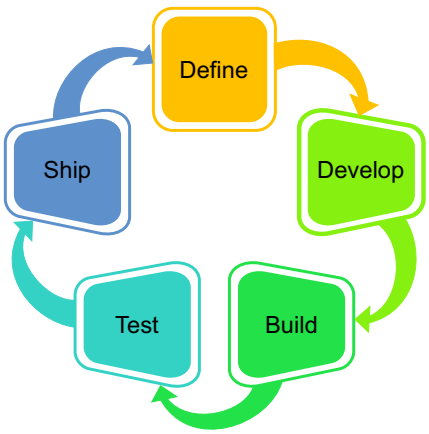
Engineering best practice

For those of you who are not software engineers or from a software engineering background, your interest in how software is developed might be extremely minimal. Why, I hear you ask, do I need to know how a developer does their job? Surely, developers know this stuff better than I do? I doubt I even understand 10 percent of it anyway!

To some extent, this is very true; developers do (should) know their stuff, and having you stick your nose in might not be welcome. However, it does help if you at least have an understanding or appreciation of how your software is created, as it can help you identify where potential issues could reside. Let's put it another way: I have an understanding and appreciation of how an internal combustion engine is put together and how it works, but I am no mechanic—far from it in fact. So, when I take my car to a mechanic for a routine service, I will question why he has had to replace all of the exhaust system because he found a fuel-injector problem—in fact, I think I would vigorously question why.

It is the same with software development and the process that surrounds it. If you're not technical in the slightest, it still helps to have an idea of how things are done. So, when you have to question why things are done in a specific way, you may be able to spot the slippery "blind them with technobabble—that'll scare them off" types.

CD is based on a premise that quality software can be defined, developed, built, tested, and shipped very quickly many times in quick succession—ideally, we're talking hours or days at the most. The following figure depicts this cycle:



A typical software delivery cycle

When looking at a typical *agile* development project, the first four steps (starting from Define) happen quite quickly and—depending on the development techniques used—might happen in a slightly different order. In a *waterfall* style project, the first four steps might take a while longer, sometimes considerably so. What both methods suffer from is their shipping step. This is normally the step that takes the time, effort, and resources—as you will have no doubt found out during the *inspect* stage (see *Chapter 2*, *No Pain, No Gain*). CD and DevOps can help reduce this pain and the time taken to ship, but you need to ensure beforehand that your engineering practices are as optimal as they can be; otherwise, you might not reap the benefits of a CD and DevOps approach.

Let's look at some fundamentals in terms of software engineering:

- Commit and merge small code changes frequently into a source control repository
- Do not make code overly complex and keep it maintainable and documented
- Avoid introducing breaking changes
- If you have automated tests, run them very frequently (ideally, continuously)
- If you have a CI solution that controls your build and test suite, run it frequently
- Use regular code reviews
- Refactor as you go

As you can see, the preceding list is not overly complex, and to most software engineers who work on modern software development projects, this list is pretty much common sense and common practice.

You might be able to speed up your software delivery process without using the preceding steps, but it will not be easy, nor will it be pretty or sustainable. What you are really trying to do is find issues as soon as possible. If you're making small changes frequently and reviewing/testing them regularly, you'll have a better chance at spotting potential bugs early on.

What it comes down to is that if you do not find software problems early on, these problems will slow you down later on and will influence the overall project. To put it another way, if there are next to no bugs when the software is delivered, releasing it should be a doddle.

Let's break these down a little further, starting with **source control**.

Source control

Source control is a must for modern collaborative software development. There are many different source control tools and solutions available (see `http://en.wikipedia.org/wiki/Comparison_of_revision_control_software` if you don't believe me). These range from commercially licensed (such as Team Foundation Server (TFS) or ClearCase) to open source ones (such as GIT, SVN, or Mercurial), so you will not struggle to find one that meets your needs and/or budget. If your code is in source control, it is versioned, it is available to anyone who has access, and it is secure.

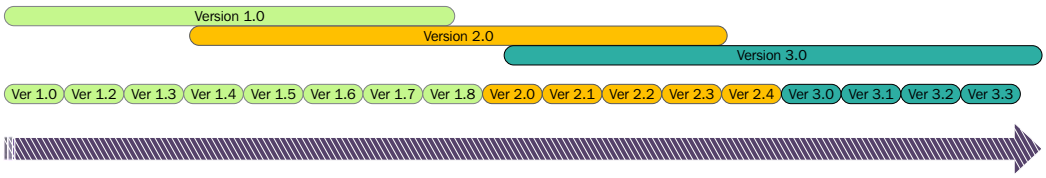
There are books and reference materials aplenty available that cover this subject in much more depth, so I will not dwell on it here. Suffice to say, if you do not have a source control solution, then implement one. Now!

It should be noted that the use of source control should not be restricted to software source code. You can (and should) utilize source control for anything that can be changed within your system as a whole. This includes things such as system configuration, start-up scripts, server configuration, network configuration, and so on. This is even more important if you are considering automated provisioning solutions (we'll cover more on this later.)

A source control solution is an essential tool for CD and DevOps adoption, as is the practice of keeping changes small and frequent.

Small, frequent, and simple changes

Keeping changes small means the impact of the change should also be small, the risks reduced, and the opportunities for change increased. This sounds overly simplistic, but this is also very true. The following diagram gives some indications on how this could look:



Large releases versus small incremental releases

What this diagram also tries to illustrate is the difference between working with large *clumps* of changes and small *chunks* of incremental changes. Typically, the large *clump* will consist of many small code changes—sometimes hundreds or thousands of lines of code—which are developed in isolation and then brought together at the last minute, tested, and merged into the core codebase. This, as you can imagine, brings with it many challenges and, if truth be told, lots of waste. Working with a small amount of change—maybe a few lines of code—doesn't bring this overhead and vastly reduces complexity and the potential for late-breaking quality issues.

Another disadvantage of working with large *clumps* of change is something I like to call **version overlap**. Typically, when development on a version has been completed, the task of bringing it all together and getting it working starts. As it doesn't normally need the whole team to do this, while the poor souls who have been given this task get on with it, the rest of the team start working in parallel on the next version. This can create a whole new set of problems—for example, the need to create more branches in source control or the fact that additional changes will be made to code that hasn't, as yet, been fully verified.

You might be thinking that there's no real point in delivering in *chunks* if you don't currently have the luxury of shipping your code frequently. To some extent, this is true; however, once you have CD and DevOps up and running, you'll be working in this mode. So, why not start getting used to it and gain some advantages beforehand?

It should be noted that this practice need not be restricted to software engineering; it is just as relevant to changes in the system operations area as well. For example, making a small, isolated tweak to server configuration is much safer and easier to control and monitor than making sweeping changes all at once. If you make small changes, you have a much better chance of seeing if the change had an impact (positive or negative) on the overall operation of the platform.

We'll now move on to how breaking changes can affect your platform.

Never break your consumer

Your platform will most probably be complex and have quite a few dependencies—this is nothing to be ashamed of and is quite normal. These dependencies can be classified as relationships between *consumers* and *providers*. The *providers* can be anything from shared libraries or core code modules to a database. The *consumers* will call/execute/send requests to the *providers* in a specific way as per a predefined API spec (sometimes called a service contract). If, for some reason, the *provider* has been changed so that what is returned is not what the *consumer* expects or differs from what was originally agreed upon, the *consumer* might behave in an unexpected way, throw an error, or simply crash.

The rule of thumb here is to avoid these situations where possible. Avoidance techniques can range from creating a dependency map of your entire platform through running impact analysis of each change to using simple peer code reviews to ensure that nothing is amiss. Do some research and see what fits best.

It should be noted that there might be exceptions when these breaking changes cannot be avoided. In such cases, you should have a strategy planned out to cater for this. An example strategy would be to accommodate side-by-side *provider* versioning, which would allow you to run more than one version of a software asset at the same time on the same platform, thus allowing *consumers* to migrate from one version of a *provider* to another over time.

Within the system operations area, the *never break your consumer* rule should also apply. For example, the software platform could be classed as a *consumer* of the server operating system (the *provider*); therefore, if you change or upgrade the operating system, you must ensure that there are no breaking changes that will cause the *consumer* to fail.

There might be times when the *consumer/provider* relationship fails, as the person or team working on the *provider* is unaware of the relationship. To overcome this, or at least to minimize the risk, open and honest peer-working practices should go some way to help.

Open and honest peer-working practices

There are many different *agile* software delivery methodologies in use today, but they all revolve around some form of collaborative and transparent ways of working, where those who are writing code regularly share their workings with others. Even the most capable engineer on the planet is human, and they can/will make mistakes—they will, of course, be reluctant to admit this.

Having a process where change is regularly reviewed by the peer group—which could include peers outside of the development team—or simply reviewed by someone sitting next to you will, among other things, help find issues early, help share knowledge, and help build relationships across the peer group. There are even tools available that will help you with this process.

Having an open, honest, and transparent peer-review process is as important within an operations team as it is within a development team. Changes made to any part of the platform run a risk, and having more than one pair of eyes to review will help reduce this risk. As with software code, there is no reason to not share system configuration changes.

One normally unforeseen advantage of peer working is that if a change fails to get through peer review, the impact on the production system is negated. It's all about failing fast rather than waiting to put something live to find it fails.

Fail fast and often

Failing fast and often might seem counterintuitive, but it's a very good ethos to work to. If a bug is created but it is not found until it has gone live—which is sometimes referred to as an escaped defect—the cost of rectifying the bug could be relatively high (it could be a completely new release). In addition, allowing defects to escape into the wild might impact your customers, your reputation, and possibly, your revenue. Finding faults early is a must.

Some engineering techniques such as **Test-driven development** (**TDD**) are based on the principle of exposing faults with software very early on in the process. If the code written fails to clear the first hurdle of tests, it goes no further.

This might sound strange—especially for the managers out there—but if bugs are found early on, you should not make a big issue of it, and you should not chastise people who have introduced the bugs (remember the no-blame culture covered in *Chapter 4*, *Culture and Behaviors*). There might be some banter around the office but nothing more. Find the problem, find out why it happened, fix it, learn from it, and move on.

To effectively find issues early on, you need to run your tests on a regular basis. The use of automation can help here.

Automated builds and tests

Finding faults early provides rapid feedback to the engineer as to whether the changes they have made actually work (or not as the case might be). You could, of course, use manual testing processes to achieve this, but this can be cumbersome, inconsistent, prone to error, and not always fully repeatable.

Implementing automation will help speed things up, keep things consistent, and, above all, provide confidence. If you are repeatedly running the same builds and tests against your software and getting the same results, it's a strong bet that the software works as expected. It is, therefore, plausible that if you change one thing (remember small and frequent changes) and the previously working builds or tests fail, there is a very good chance that the change has broken something.

Test data can be a bit of a thorny issue and can cause more problems than it solves. A good rule of thumb would be to create and tear down the test data you need when the test is being run; this way, the outcome of the test will not be impacted by pre-existing data, which might itself be faulty.

How you go about choosing the best approach, tools, and techniques for automation can be daunting, but applying the KISS rule will help; start small, focusing on your major pain points, and then evolve as your confidence grows. Suffice to say that the investment in automation will reap rewards. One such reward is the ability to use continuous integration.

Continuous Integration

Continuous Integration (**CI**) is a tried and tested method of ensuring that a given software asset builds correctly and *plays nicely* with the rest of the platform. The keyword here is *continuous*, which, as the name implies, is as frequent as possible (ideally, upon each change). Just like the aforementioned source control solutions, there are quite a few CI tools available, from the commercially licensed (such as Bamboo, Go, or TeamCity) to open source ones (such as Jenkins, Hudson, or CruiseControl). A pretty comprehensive list is available at `http://en.wikipedia.org/wiki/Comparison_of_continuous_integration_software`—which also provides a good indication of which CI solutions play nicely with each of the source control solutions currently available.

CI systems are basically software solutions that orchestrate the execution of your automated scripts when certain events occur, for example, when you commit a change to source control. These *CI jobs* contain a list of activities that need to be run in quick succession; for example, get the latest version of source from source control, compile to an executable, deploy the binary to a test environment, get the automated tests from source control, run them, and capture the results.

If all is well, the CI job completes and reports a success. If it fails, it reports this fact and provides detailed feedback as to why it failed. Each time you run a given CI job, a complete audit trail is written for you to go back and compare results.

CI tools can be quite powerful, and you can build in simple logic to control the process—for example, if all of the automated tests pass, the CI tool can add the executable to your binary repository, or if something fails, it can e-mail the results to the engineering team. You can even build dashboards or radiators to provide an instant and easy-to-understand visual representation of the results.

CI solutions are a must for CD. If you are building and testing your software changes on a frequent basis, you can ship frequently. Using a CI solution will also help in building the DevOps relationships, as the Ops half of the relationship will be able to see proof that builds and tests have been successful. They could also be heavily involved in defining and creating some of the dashboards.

Your CI solution will help you build and test your software to ensure it is fit to ship. You now need to ensure you can take into account how to move this built software towards the production environment. Before we do this, there is one more thing to take into account.

Using the same binary across all environments

When a software asset is ready to be shipped, it has normally been built/compiled into an executable. To ensure that the software functions in the production environment as it did in your development and/or test environment(s), you need to ensure that this self-same unchanged binary is used. This might sound like obvious common sense, but sometimes, this is overlooked or simply ignored.

There might be issues related to this which are not obvious, for example, if the software needs access to certain secure data (such as credentials to connect to a database). If you have traditionally *baked* this into the binary at build time (or worse still, hardcoded this in source), you will need to change this practice. This might well require some additional development work and should not be taken lightly.

Let's now look at the optimal use of environments for various steps within your development process.

How many environments are enough?

How many environments you need depends on your ways of working, your engineering setup, and, of course, your platform. Suffice to say that you should not go overboard. There might be a temptation to have many environments set up for different scenarios: development, functional testing, user-acceptance testing, and performance testing. If you have the ability to ensure that all the environments can be kept up to date (including data) and you can easily deploy to them, then this might be viable. The reality is that having too many environments is counterproductive and can cause far too much noise and overhead.

The ideal number is two—one for development and one for production. This might sound like an accident waiting to happen, but many small businesses manage fine with such a setup.

For your development environment, you could look at virtualization on the desktop (using a tool such as Vagrant or Docker) where you can pretty much spin up a virtual copy of your production environment on a developer's workstation (on the presumption that there's enough horse power and storage available).

It's a fact of life that as a business grows, so does the need to be risk averse. This normally means more checks and balances are needed, which lead to burdensome processes and inevitably more and more environments being created to allow for these processes to work. It need not be this cumbersome. My advice is to be very prudent and somewhat ruthless when it comes to the number of environments you need (need rather than want). Get the Dev and Ops team together with the team(s) that look after change and risk management to ensure that there is a middle ground that allows for speed of delivery while, at the same time, allows for a reduced/managed risk.

When your CD and DevOps adoption has matured, you will be shipping code very quickly. Try to imagine how much overhead managing and maintaining multiple environments will be and how it could slow you down.

One thing that might sway this decision is having the ability to develop against a production-like environment.

Developing against a production-like environment

As we've mentioned previously, there are many ways to ensure that a software change will play nicely when it is deployed to the production platform. By far the easiest is to actually develop and test against an environment that is as close to your production environment as possible.

The utopia would, of course, be to develop and test against the production environment, but this is very risky, and the possibility that you could cause outage—albeit inadvertently—is quite high. Therefore, having an environment that closely resembles the production environment is the next best thing.

As mentioned previously, there are tools available to allow developers (or anyone who wants to) to spin up a virtual version of your production environment.

With this kind of environment in place, you can then develop in isolation, but be more confident that the changes that are being made will play nicely when you ship the software to the production environment. There is one thing you will need to take into account with this approach—which is also true of any development or test environment—that being realistic like-live data. You need to consider how you can create a realistic subset of your production data. Alternatively, you could consider allowing access to production databases (as long as this access is secure). I suggest you work with your **database administrator** (**DBA**) and **Security Operations** (**SecOps**) team to work out the best approach.

You are starting to get all of the building blocks in place to realize your goal. There are still, however, few other hurdles to get over, one of them being how you seamlessly move the fully built and tested software asset through to production. Here is where CD tooling comes into play.

CD tooling

You could consider CD tools as being the natural evolution of the aforementioned CI tooling. Instead of controlling and orchestrating the build and test process, CD tools control and orchestrate the act of deploying the built software components to the various environments you have. Currently, there doesn't seem to be a standard set of standalone CD tools. Instead, many vendors and tool creators have extended their products to include this functionally.

Before committing to a tool or solution, you should consider some of the following questions:

- Can it deploy the same binary to multiple environments?
- Can it seamlessly access the binary and source repositories?
- Can it remotely invoke and control the installation process on the server that it is being deployed to?
- Is it capable of deploying database changes?
- Does it have the functionality to allow for queuing up of releases?
- Does it contain an audit of what has been deployed, when, and by whom?
- Is it secure?
- Can it interact with the infrastructure to allow for no-downtime deployments?
- Can it/could it orchestrate automated infrastructure provisioning?
- Can it be extended to interact with other systems and solutions such as e-mail and change-management, issue-management, and project-management solutions?
- Does it have simple and easy-to-understand dashboards that can be displayed on big screens around the office?
- Can it interact with and/or orchestrate the CI solution?
- Will it grow with our needs?
- Is it simple enough for anyone and everyone to use?

If you manage to find a tool (or collection of tools) that answers most/all of these questions, then congratulations! If not, then you should seriously consider building some of your own tools (or possibly bolting some existing tools together). After all, you want to enhance your process with tools rather than restrict your process because of the tools you chose.

It should be noted that deployment of database changes via CD tooling can be a very complex thing to do compared to deploying software. If this is a must-have requirement, you should take time and run some trials before committing to a tool or approach.

Let's spend some time digging into a couple of the considerations listed earlier, starting with automated provisioning.

Automated provisioning

If you are lucky enough to have a platform (or have re-engineered your platform) that can run on virtual infrastructure, then you can consider automated provisioning as part of the deployment process.

Automated provisional tools are available for non-virtualized (read physical) environments and platforms, but these can be overly complex and costly. If you can utilize virtualization, you should.

Automated provisioning is nothing new. The likes of Amazon and Google have been providing their *cloud-based servers* for a while now, and you have been able to provision what servers you want when you need them (for a price).

Having automated provisioning as a step within the deployment process is something that is extremely useful and powerful. It should be noted that it can also be quite complex and, at times, painful to implement—unless you know what you're doing.

As is normal within the IT industry, there are many buzzwords floating around to add to this complexity—**Infrastructure-as-a-Service** (**IaaS**) and **Platform-as-a-Service** (**PaaS**) being the most common ones. Simply put, automated provisioning solutions allow you to programmatically talk to a system; provide it with a recipe that contains things such as server spec, operating system, configuration, and so on; and it spits out a server at the other end.

In addition to IaaS and PaaS, there's another widely used buzzword—**Infrastructure-as-Code** (**IaC**). In simple terms; this is, as the name implies, code written in what could be (and pretty much is) classed as a development language that defines the recipe and the process of provisioning.

Infrastructure-as-code is an area where the notional line between Dev and Ops becomes blurred, and most of the aforementioned engineering practices become as relevant to your average Oscars as they do for your Devinas.

With automated provisioning, getting your software up and running can be relatively simple. You have the fully tested software asset, and you have the recipe for your environment/server configuration, so the act of deployment could be as simple as shown here:

OK, so there is a little more to it than that, but to all intents and purposes, if you have the ability to do this, then there is no reason you should not consider it. If you wish to go down this route, it is vitally important that your CD tooling allows for it. One other benefit of automated provisioning at deployment time is that it can help a great deal in terms of no-downtime deployments.

No-downtime deployments

One of the things that come with large releases of software (legacy or otherwise) is the unforgivable need to take some or all of your production platform offline while the release happens. Yes, I did say unforgivable, because this is exactly what it is. It is also wholly avoidable.

If you are operating a real-time online service, you can bet a pretty penny that your customers will not take kindly to not being able to access your system (or more importantly, their data) for a few hours so that you can upgrade some parts of it. There is also a very strong possibility that they will look upon this with distrust as they'll be pretty sure something will go wrong once it's up and running again. It will and you will then be in *damage-limitation* mode to keep them happy. They might even shop around to find a competitor who does not have downtime.

OK, so this is a bit on the dark and negative side, but this is the reality, even more so with today's social media and viral ways of spreading bad news—some say that bad news travels faster than anything else known to man. The last thing you need is bad news generated because of a release; this will knock your confidence, tarnish your reputation, and erode any trust you had built up within the business. Release-related incidents can and will happen, so adding insult to injury is not ideal.

There are many simple things that can be done to remove the need for downtime deployments, some of which we have already covered:

- Ensure the *never break your consumer* rule is followed religiously
- Ensure your changes are small and discrete
- If possible, implement automated provisioning and integrate this as part of your CD tooling
- Implement load balancers and have the CD tooling orchestrate servers in and out of the pool during the deployment
- If you cannot avoid implementing breaking changes, then do so gradually rather than with a big bang

There are, of course, many more things that you might be aware of or can find information on elsewhere, but suffice to say that if you ever have to take your platform offline to release software, something is fundamentally wrong.

One thing to point out, which might not be obvious, is that it's not just the production environment that should have maximum uptime. Any environment that you rely on for your development, testing, and CD should be treated the same. If the like-live environment is down, how are you going to develop? If your CI environment is down, how are you going to integrate and test? The same rules should apply across the board—without exception.

To reuse a metaphor from an earlier chapter, there is an elephant in the room that we've been skirting around. This elephant has fast become a must-have big-ticket item for most software businesses and is seen as synonymous with the adoption of CD and DevOps. This elephant is the cloud.

The cloud

As mentioned earlier, delivering software using cloud solutions and technologies has been around for a while; however, the vast majority of businesses around the globe have not taken full advantage of this phenomenon as yet —especially in their production environments. The uptake of cloud solutions and technologies within the enterprise space, for example, is growing, but the numbers small and uptake are still very slow. That's not to say it can't be done; it's just a bigger leap for some well-established and old-school businesses.

One of the major advantages of using cloud-based infrastructure is the plethora of mature and proven tools, technologies, and techniques that have emerged in recent years. You can pick them up with relative ease and—being mostly open source—without heavy investment. There's also a fast-growing and vastly experienced community globally available that can help you at every stage.

Adopting cloud-based technologies can help with accelerating your CD and DevOps adoption. The aforementioned plethora of tools will allow you to fast track some areas of your adoption—for example, you would be able to let engineers spin up new servers with relative ease to simply try some experiments with a new CI module or see whether an OS upgrade has any impact on the software platform.

In a *quid pro quo* kind of way, adopting DevOps ways of working can also help accelerate the adoption of cloud-based solutions and technologies. You'll need closely-knit Dev and Ops teams that can work seamlessly and collaboratively to implement, set up, and manage your virtual IaC-based environment. They will also need to work closely together to monitor everything that is happening in their now globally distributed platform.

Before you go all guns, there are, however, a few caveats you should take into account when it comes to cloud adoption. Here are a few examples:

- A major hurdle for some is the simple fact that the skills and experience needed to effectively use these tools, technologies, and techniques are more attuned to developers than traditional system operators—more Dev than Ops. There is, therefore, something of a learning curve and possibly some skillset realignments that are required.
- Scale can also bring new problems—imagine trying to deploy a software asset to thousands of servers without a Dev and Ops team working cohesively with a shared toolset in a highly collaborative way.

- There is also the age-old issue of security. It's hard enough for your average SecOps person to ensure adherence to regulations and data security when your data lives on servers that are sitting in a locked-down server room—imagine their reaction to *we're moving it all to the cloud*. Without having open, honest, and trust-based relationships in place, you're going to struggle to help these people overcome their initial shock and feeling of dread.

I'm pretty sure you would be able to add to this list; however, you will find that the proven advantages of adopting cloud-based solutions and technologies will outweigh the caveats.

Previously, we covered open and honest ways of working as part of engineering best practices. Openness and honesty are just as important when it comes to CD. A good way of providing this level of transparency is to monitor everything and have it available to all.

Monitoring

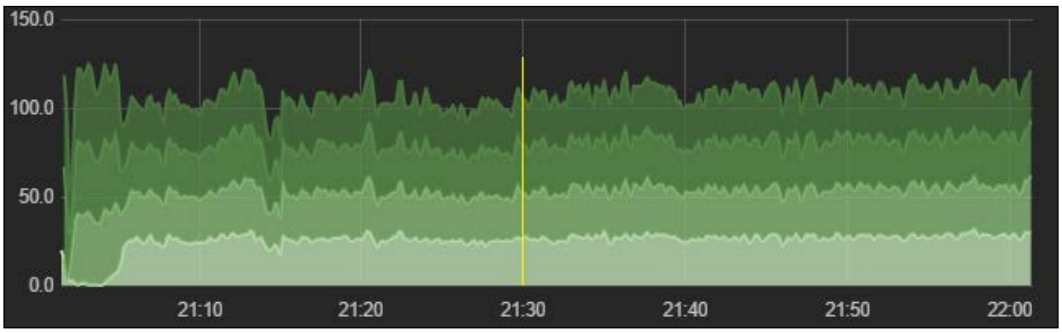
One of the most important ways to ensure whether CD and DevOps is working is to monitor, monitor, and then monitor some more. If all of the environments used within the CD process are constantly being observed, then the impact of any change (big or small) is easy to see—in other words, there should be no hidden surprises.

If you have good coverage in terms of monitoring, you have much more transparency across the board. There is no reason why monitoring should be restricted to the operations teams; everyone in the business should be able to see and understand how any environment—especially the production platform—is performing and what it is doing.

There are plenty of monitoring tools available, but it can be quite difficult to get a single view that is consistent and meaningful. For example, something like Nagios is a pretty good tool for monitoring infrastructure and servers, Graphite is pretty good at collecting application metrics, and Logstash is pretty good at collecting and analyzing server logfiles. Unless you can tie them all together into a single coherent view, things will look disjointed. Ideally, you should try and aggregate the data from these tools—or at least try and integrate them—and present a unified view of how the production platform (or any environment for that matter) is coping and functioning. You will be surprised how much extremely valuable data you can get and how it can direct your development work, as the engineers can see exactly how their software or infrastructure is behaving in real time with real users.

Monitoring is a must for CD and DevOps. As change is being made to production (software or infrastructure), both the Dev and Ops sides of the DevOps partnership can see what is going on and assist when/if problems occur.

Another less-obvious positive that monitoring can bring you is proof that CD is *not* having an adverse impact on the environment to which you are deploying. Let's assume that you are using some *graph over time* monitoring solution that is recording and graphing things such as network throughput, server load, and application performance. If you, then, somehow get your CD tool(s) to record a *spike* or a *marker* to the graph when a deployment takes place, you can then visually see when the deployment took place and what impact, if any, it had. If the other metrics show little or no change during/after the deployment, you can be pretty confident that there has been no impact. The following figure shows this (the deployment took place at 21:30):



An example graph over time showing a deployment

Up until this point, we have mainly focused on technical solutions. These solutions might help to provide you with much of what you need in your toolbox. However, there is still room for simple manual processes, which complement the technical solutions.

When a simple manual process is also an effective tool

Even if you have enough tooling to shake a stick at, you will no doubt have some small and niggling challenges that cannot be overcome with tooling and automation alone. To be honest, tooling and automation can be overkill in some respects and can actually create barriers between certain parts of the organization you are trying so hard to bring together—here, I am talking about the Dev and Ops partnership that forms DevOps.

If tooling and automation completely negate the need for human interaction and discussion, you might well end up back where you started. You might also find that it is almost impossible to automate your way out of a simple problem.

Let's take, for example, the thorny issue of dependency management. As a software platform matures, many interdependencies will form. If you are deploying your code using a CD process, these many interdependencies become ever-moving targets where components are being developed and deployed at different rates. You can try to capture this within your CI process, but something somewhere might be missed, and you could end up inadvertently bringing down the entire platform because component B was deployed before component A.

You can try to map this out and build into the tooling rules to restrict or at least minimize these moving targets, but the rules might end up more complex than the original dependencies themselves. Alternatively, you could simply agree on a process whereby only one change happens at any given point in time. To feed into this, you can implement a simple queuing mechanism written on a whiteboard and reviewed regularly by all of the engineering and operations teams.

This approach worked extremely well for ACME systems. The following is what they did:

- They obtained blanket agreement from everyone that only one change would go through to production at any given point in time. They called this a *deployment transaction*.
- None of the CD tooling was changed to have this restriction built in; they simply relied on common sense and collaborative ways of working.
- To highlight the fact that someone was making a change to production (either a deployment or operational change), that person held the *production environment token*, which was in the form of a plush toy animal and was given the name *the deployment badger*. If you had the deployment badger, you were changing production.
- They implemented a simple prioritized queue system using a whiteboard. Each morning, whoever wanted to make a deployment would come along to the deployment stand-up where everyone agreed the order in which deployments (or changes) would be made that day.
- Screens were installed throughout the office (not just the Dev and Ops areas), showing a real-time dashboard of what was going on.

All very simple, but what this gave ACME systems was a way to overcome dependency hell (for example, if they could only change one thing at a time, there was an implied logical order of which change went before another) and built a sense of collaboration throughout all the teams involved.

Other very simple manual solutions you can use could include the following:

- Use collaborative tools for real-time communication between everyone (that is, **internet relay chat** (**IRC**), chat rooms, or similar) and integrate this into your CD tooling so that deployments are announced and can be followed by all
- If your management is uneasy about having developers deploy to production without involving the operations team, set up a workstation within the operations area, call it the *deployment station*, and make sure that's the only workstation from where live deployments can be run from
- If instant rollback is needed should a deployment fail, consider simple ways of rolling back, such as deploying the previous version of the component using the CD tooling
- Consistently inspect and adapt through regular retrospectives to see what is working and what is not

As you can tell, it's not all about technical solutions. If simple manual processes or tweaks to the ways of working are sufficient, then why bother trying to automate them?

And so ends the lesson—for now. Let's recap what we have covered throughout this chapter.

Summary

As stated at the beginning of this chapter, there is a lot to cover and take in. Some of it is relevant to you now, and some of it will be relevant for the future. All things considered, you need to ensure that you have the technical building blocks in place. You need to ensure that you are using engineering best practice, have the necessary tools and solutions in place, work in small *chunks* of change rather than big *clumps*, consider how many environments you actually need, consider simple manual processes over technology where they fit best, and monitor, monitor, and then monitor some more.

As you can see, there is quite a bit that needs to be done. It's not all technical either; simply convincing people to adopt and use the tools and processes you implement is no small task. In *Chapter 6*, *Hurdles Along the Way*, we'll look at some of the other hurdles and challenges you'll face.

6
Hurdles Along the Way

Up until now, we have been focusing on the core tools and techniques you'll need in your toolbox to successfully implement and adopt CD and DevOps. Along the way, we looked at a few of the hurdles you'll have to get over. We'll now look at some of these potential hurdles in more detail and the ways to overcome them or at least minimize the impact of them so that you can drive forward with your goal and vision.

What follows is by no means an exhaustive list; however, there is a high probability that you'll encounter at least one or two of these problems along the way. The main thing that you need to do is be aware that there will be the occasional storm throughout your journey. You need to understand how you can steer your way around or through it and ensure that it doesn't ground you or completely run the implementation onto the rocks—to use a nautical analogy for some reason.

What are the potential issues you need to look out for?

Depending on your culture, environment, ways of working, and business maturity, there might be more potential hurdles than you can shake a stick at. Hopefully, this will not be the case, and you will have a nice, smooth implementation, but just in case, let's go through some of the more obvious potential hurdles. What follows are some example hurdles you may encounter:

- Individuals who just don't see why things have to change and/or simply don't want to change how things are
- Individuals who want things to go quicker and are impatient for change
- The way people react to change at an emotional level

- A lack of external understanding or visibility of what you are trying to achieve might throw a spanner in the works when business priorities change
- Red tape and heavyweight corporate processes
- Geographically diverse teams
- Unforeseen issues with the tooling chosen for the tool kit (technical and nontechnical)
- Recruitment

The list could be much longer, but there's only so much space in this book. Let's therefore focus on more obvious potential issues that could if left unchecked run the implementation of CD and DevOps into shallow waters or, worse still, aground. We'll start by focusing on individuals and how they can have an impact, both negative and positive, on your vision and goal.

Dissenters in the ranks

Although the word **dissenters** is a rather powerful one to use, it is quite representative of what can happen should individuals decide what you are doing doesn't fit with their view of the world.

As with anything new, some people will be uncomfortable, and how they react depends on many things, but there's a strong chance that, you will have some individuals who decide that they are against what you are doing. The whys and wherefores can be examined and analyzed to the *n*th degree, but there is something very important for you to realize: if one or two individuals are loud enough, they can redirect your attention from your vision and goal. This is exactly what you don't want to happen.

This is nothing new. If you look back at the early days of *agile* adoption, there are plenty of examples of this phenomenon. The individuals involved in the adoption of *agile* within an organization broadly fall into three types:

- A small number of **innovators** trailblazing the way
- A larger number of **followers** who are either interested in this new way of doing things or can see the benefits and have decided to move in the direction that the innovators are going
- Finally, the **laggards** who are undecided or not convinced that it's the right direction to go.

The general consensus is that effort and attention should be focused on the innovators and followers as this makes up the majority of the individuals involved. The followers who are moving up the curve towards the innovators need some help to get over the crest and over the other side, so more attention is given to them. To focus on the laggards would take too much attention away from the majority, so the painful truth is that they either shape up or ship out—even if they're senior managers. This might seem rather brutal, but this approach has worked for a good number of years, so there must be something in it.

So, let's consider our dissenters or laggards in terms of our implementation; what should you do? As previously pointed out, if they are loud enough, they can make enough noise to disrupt things, but not for long. If the majority of the organization has bought into what you are doing—don't forget that you are executing a plan based on their input and suggestions—they will not easily become distracted; therefore, you should not become distracted. If you have managed to build up a good network across the business, use this network to reduce the noise and, if possible, convert the laggards into followers.

If these laggards are in managerial positions, this might make things more difficult—especially if they are good at playing the political games that go on in any business—however, they will be fighting a losing battle as the majority will be behind you (because you are delivering something they have asked for). You just need to be diligent and stick to what you need to do. You will have your eyes peeled and your ear to the ground, so you should be able to tell when trouble is brewing, and you can divert a small amount of effort to addressing this and stop it from becoming a major issue. The *addressing this* part can be in the form of a simple non-confrontational face-to-face discussion with the potential troublemaker over a coffee—this way, they are being listened to, and you have an idea of what the noise is all about. As a last resort, a face-to-face discussion with their boss might do the trick. Don't resort to e-mail tennis!

All in all, you should try wherever possible to deal with dissenters as you would the naughty child in the classroom; don't let them spoil things for everyone; don't give them all of the attention; and use a calm, measured approach. After a while, people will stop listening to them or get bored with what they are saying (especially if it's not very constructive).

Let's look a couple of examples that could potentially fuel the voice of the dissenters.

No news is no news

Something that could increase the risk of dissenters spoiling the party is a lack of visible progress in terms of CD and DevOps implementation. It might be that you're busy with a complex process change or developing tooling, and there is a lull in visible activity. If you have individuals within your organization who are very driven and delivery focused, they might take this lull as a sign of the implementation faltering, or they may even think that you're finished.

As we mentioned previously, being highly visible, even if there's not a vast amount going on, is very important. If people can see that progress is being made, they will continue to follow. If there is a period of perceived inaction, the followers might not know which way you are heading and might start taking notice of the dissenting voices. Any form of communication and/or progress update can help stop this from happening—even if there's not a vast amount to report, the act of communication indicates that you are still there and still progressing towards the goal.

The anti-agile brigade

In *Chapter 5*, *Approaches, Tools, and Techniques*, we looked at engineering best practice, which, if truth be told, is simply based on modern *agile* engineering techniques. You might find that some of the dissenting voices belong to the old-school engineers who have an aversion to anything *agile*. Maybe they have been working in the same mode for the past 20 years and believe that they are exempt from this *new fad*. Maybe they simply don't understand it or are afraid of working in such a way. Whatever the reasons, you need to be mindful of the fact that they will cause some noise. If they are relatively senior and/or well respected, you should tread carefully as you don't want to end up with a power struggle.

Focus your time and energy on the followers and innovators and see if you can get some of them to work with the old-school laggards and ease them in gently.

You should also consider working with their managers and looking into specific training or incentives to help ease their move from the dark side—without resorting to or being seen to resort to bribery. What you need is more innovators and followers than laggards.

If your entire engineering team's ways of working are based on old-school *waterfall* delivery practices, your hurdles are going to be much larger. In reality, CD and DevOps do not play nicely with traditional *waterfall* software delivery; therefore, you are going to have to address this problem with some urgency. You should consider investing time, effort, and money in training, sending people to relevant conferences or local meet-ups—again, focus on those individuals who have the potential to become your innovators, as they can help drive forward the adoption of *agile* engineering practices.

We briefly covered the fact that some people will be uncomfortable with change, and they might react in unexpected ways. We'll now look at how change can impact individuals in different ways and what you need to be aware of.

The transition curve

Let's get one thing out in the open—and this is important—you need to recognize and accept that the identification of a problem and subsequent removal of it can be quite a big change. You have been working with the business to identify a problem, and you are now working to remove it. This is change, pure and simple.

Earlier in the book, we stated that the brave men and women of ACME systems who helped implement the DevOps and CD ways of working were a catalyst for change. This wording was intentional as change did come about for the ACME systems team—a very big change as it turned out. The implementation of CD and DevOps and its impact on individuals should not be taken lightly, even if they originally thought it was the best thing since sliced bread.

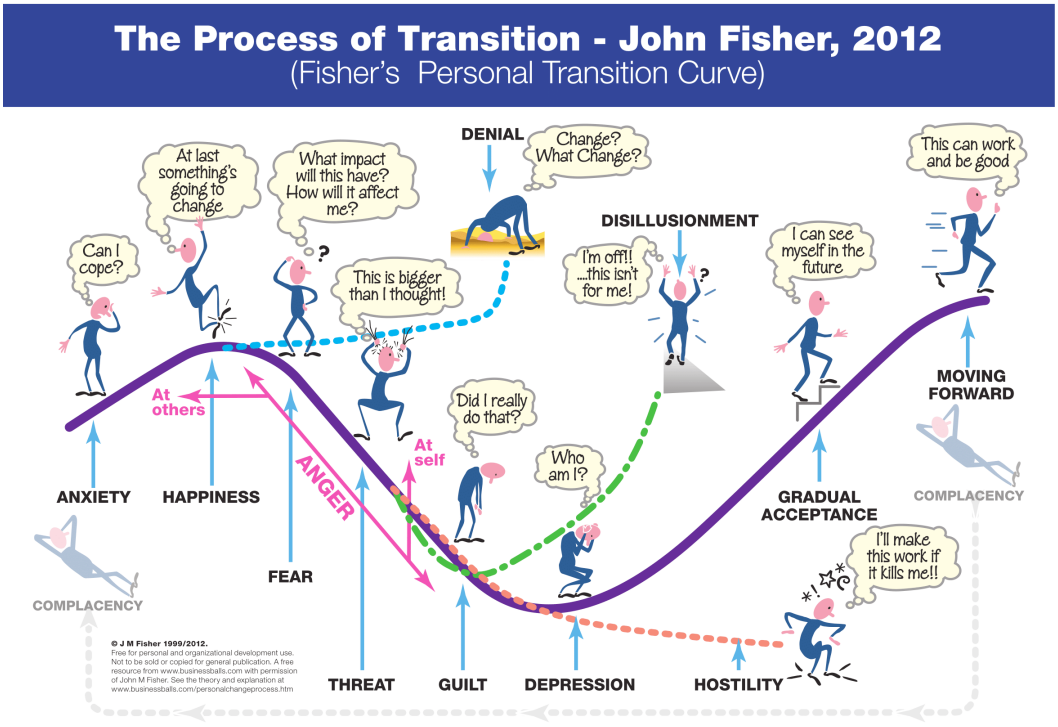
Those of you who have been in, or are currently in, management or leadership roles should understand that change can be seen as both positive and negative, and sometimes, it can be taken very personally—especially where a change to a business impacts individuals and their current roles within it. Let's look at some fundamentals in relation to how humans deal with change.

Any change, large or small, work related or not, can impact people in different ways. Some people welcome change, some are not fazed by it and accept it, some are downright hostile and see a change as something personal. More importantly, some people are all three. If we are mindful of these facts before we implement change, we will have a clearer idea of what challenges to overcome during the implementation to ensure that it is successful.

There has been much research on this subject, and many papers have been published by learned men and women over the years. I don't suggest for one minute that I know all there is to know on this subject, but there is some degree of common sense required when it comes to change—or transition as it is sometimes called—and there are some very simple and understandable traits to take into account.

One of my preferred ways to visualize and understand the impact of change is something called the change or transition curve. This depicts the stages an individual will go through as change/transition is being implemented.

The following diagram is a good example of a change/transition curve:



John Fisher's personal transition curve—the stages of personal change

John Fisher's personal transition curve diagram can be found at `http://www.businessballs.com/freepdfmaterials/fisher-transition-curve-2012bb.pdf`.

You can clearly see that as change is being planned, discussed, or implemented, people will go through several stages. We will not go through each stage in detail (you can read through this at your leisure at `http://www.businessballs.com/personalchangeprocess.htm`); however, there are a few nuggets of information that are very pertinent when looking at implementing CD and DevOps. They are as follows:

- People might go through this curve many times—even at the very early stages of change
- Everyone is different, and the speed at which they go through the curve is unique to the individual
- You and those enlightened few around you will go through this curve
- Those who do not / cannot come out of the dip might need more help, guidance, and leadership
- Even if someone is quiet and doesn't seem fazed, they will inevitably be sat at some stage in the curve—it's not just the vocal ones to look out for

The long and short of it is that individuals are just that; they will be laggards, followers, or innovators, and they will also be somewhere along the change curve. The leaders and managers within your organization need to be very mindful of this and ensure that people are being looked after. You also need to be mindful of this, not least because this will also apply to you, as it might give some indication as to why certain individuals act in one way at the beginning, yet they change their approach as you go through the execution of the plan and vision.

At a personal and emotional level, change is good and bad, exiting and scary, challenging and daunting, welcomed and avoided. It all depends on how you, as an individual, feel at any given point in time. CD and DevOps is potentially a very big change; therefore, emotions will play a large part. If you are aware of this and ensure that you look for the signs and react accordingly, you will have a much better time of it. Ignore this, and you will have one hell of a battle on your hands.

On this light note, we'll move on to the subject of what to do about those people within your organization who are not involved in your journey or might not even be aware that it is ongoing. We'll call them *the outsiders*.

The outsiders

The percentage of those involved with the implementation of CD and DevOps will largely depend on the overall size of your organization. If you are a startup, the chances are that everyone within the organization will be involved. If you are a **small to medium enterprise** (**SME**), there is a good chance that not everyone within your organization will be involved. If you are working within a corporate business, the percentage of those actively involved will be smaller than those who are not.

The following diagram illustrates how a typical corporation is made up and where you and your team sit within it:

The further out from the inner circle, the greater the possibility that there is ignorance about what you are doing and why

Those sitting outside of the circle of active involvement will have little/no idea of what is going on and could—through this lack of knowledge—put hurdles in the way of your progress. This is nothing new and does not specifically apply to the implementation of CD and DevOps; this is a reality for any specialized project. Let's take a look at ACME systems and see how this situation impacted their implementation.

During the version 2.0 stage of their evolution, ACME systems became part of a large corporate. They ended up as a satellite office—the corporate HQ being overseas—and on the whole, were left to their own devices. They beavered away for a while and started to examine and implement CD and DevOps. They were doing so, when viewed at a global corporate level, in isolation. Yes, they were making far-reaching and dramatic changes to the ACME systems organization, but they were a small cog in a very big wheel. No one outside of the ACME systems offices had much visibility or in-depth knowledge of what was going on.

As a consequence, when a new far-reaching corporate strategic plan was announced, little or no consideration was given to what ACME systems were up to; no one making the decisions really knew. As a result, the progress of the CD and DevOps implementation was impacted.

In the case of ACME systems, the impact turned out to be positive in respect to the CD and DevOps implementation and actually provided an additional boost—you might not be so lucky. If you experience wide-reaching changes during your journey and people are ignorant of what you're doing, your story might not end so well. Bear this in mind.

The moral of the story is this: not only should you keep an eye on what is happening close to home, but you should also keep an eye on what is happening in the wider organization. We've already looked at how important it is to communicate what you are doing and to be highly visible. This communication and visibility should not be restricted to those immediately involved in the CD and DevOps implementation; you should try to make as many people aware as possible. If you are working within a corporate environment, you will, no doubt, have some sort of internal communications team who publish regular news articles to your corporate intranet or newsletter. Get in touch with these people and get them to run a story on what you are doing. A good bit of PR will help your cause and widen the circle of knowledge.

This might seem like quite a lot of work for little gain, but you might be surprised how much benefit it can bring. For example, let's imaging that you write an article on CD and DevOps, get it published, and it is read by your CEO or SVP who then decides to visit and see what all the fuss is about. This is a major moral boost and good PR. Not only that, but it can help with your management dissenters—if they see the high-ups recognizing what you are doing as a positive thing, they might (will) reconsider their position.

We're primarily considering outsiders as individuals outside of your immediate sphere of influence that are ignorant of what you are doing and where you're heading. You might have others who are well aware, but are either restricted by or hiding behind corporate red tape. Let's spend some time looking at this potential hurdle and what can be done to overcome it.

Corporate guidelines, red tape, and standards

The size and scale of this potential hurdle is dependent on the size and scale of your organization and the market in which you operate. For example, you might work within the service sector and have commercial obligations to meet certain **service-level agreements** (**SLAs**), or you might work within a financial institution and have regulatory and legal guidelines to adhere to. Whatever the industry, there's a strong bet that you will be hampered in some way by things and people outside of your control. This, as they say, comes with the territory.

To overcome this hurdle or, at the very least, minimize the impact, you need to work closely with those setting and/or policing the rules to see what wriggle room you have. Build rapport, entice them with coffee and doughnuts, get to know them, and understand what drives them and what constraints they have to work with. Eventually, you'll start to build a picture of what constraints you have and which of these could have the biggest impact on your plan and vision. You might also find that things aren't as black and white as first thought. For example, you might find that some of the rules and guidelines set in place are overkill and have only been implemented in their current form, because it was easier, quicker, or safer to stick to what it said in a book than it was to refine to fit the business needs.

Fundamentally, the need for such rules, guidelines, and policies revolves around change management and auditability. In simple terms, they offer a safety gate and a way to ascertain what has been changed by whom, should problems occur.

One hurdle that isn't immediately obvious is that those managing or policing the rules, guidelines, and policies might well consider CD and DevOps to be incompatible with their ways of working. If you think about it, they'll be imagining software flying through the door at a rate of knots. This might be a scary vision for them.

They might also consider CD and DevOps as a threat and be overly defensive. This might be due to the fact that during the *inspect* stage (see *Chapter 2*, *No Pain, No Gain*), their organization/department had been highlighted as an area of waste (I would put money on it), and as such, they might not be totally willing to embrace the change you are bringing. It might even be the case that they simply don't know what they can change without breaking a rule or corporate policy. Use your new-found rapport and work with these people and help them understand what CD and DevOps is all about and help them research what parts of their process they can change to accommodate. Do not simply ignore them and break the rules, as this will catch up with you later down the road and could completely derail you. Open, honest, and courageous dialogue is the key (see *Chapter 4*, *Culture and Behaviors*).

That said, open and honest dialogue might be hindered by geography, so let's look at how we can address this.

Geographically diverse teams

We previously touched upon the subject of setting up an open and honest physical environment to help enforce open, honest, and collaborative ways of work. This is all well and good if the teams are collocated; however, trying to recreate this with geographically diverse teams can be a tricky problem to solve.

It all depends on the time-zone differences and, to some extent, cultural differences. Not having a physical presence is always a barrier—unless you have perfected the art of matter teleportation; however, there are a few things that you could look at to overcome this:

- Ensure local team(s) have regular (ideally daily) teleconference calls with the remote team(s)—even if it's nothing more than to say good morning.
- If you're using *scrum* (or a similar methodology) and decide to have a daily *scrum* of *scrums*, get the remote teams(s) dialed in as well—even if you call them on your cell phone and have them on speakerphone.
- Time zones can play havoc with daily stand-ups (from experience, these normally happen first thing in the morning), so try and be creative around the schedule (don't forget to accommodate changes to clocks throughout the year).
- We all have access to some form of video conferencing—be that using a corporate teleconferencing system or something as simple as Skype (or similar). You could set this up within your office space (rather than hidden away in some meeting room) and use it like an *always-on* virtual wall / window that allows team members to simply walk up and have a face-to-face conversation, as if they were in the same room/city/country.
- If your budget allows, try and get people physically swapped across the offices either via secondments or short-term project placements.
- Don't rely on e-mails for discussions; instead, invest in real-time collaborative tools and encourage their use.

Cultural differences were previously mentioned as a potential problem. This should not be taken lightly. The reality is that in some parts of the world, the culture is far and removed from the fast and loose western culture where everyone has a voice and isn't afraid to use it. Instilling openness, honesty, and transparency might be more difficult for some, and you should be mindful of this. Work with the local HR or management, explain what you're trying to do, and see what they can do to help with this.

We'll now look at what you should do if you encounter failure during the execution of your goal and vision.

Failure during evolution

As you go along your journey, things will occasionally go wrong—this is inevitable and is nothing to be afraid or ashamed of. There might be situations that you didn't foresee, or there might be hidden steps in an existing process that were not surfaced during the *inspect* stage (see *Chapter 2*, *No Pain, No Gain*). It might even be as simple as a problem within the chosen toolset, which isn't doing what you had hoped it would or is simply buggy.

Your natural reaction might be to hide such failures or at least not broadcast the fact that a failure has occurred. This is not a wise thing to do. You are working hard to instill a sense of openness and honesty, so the worst thing you can do is the exact opposite.

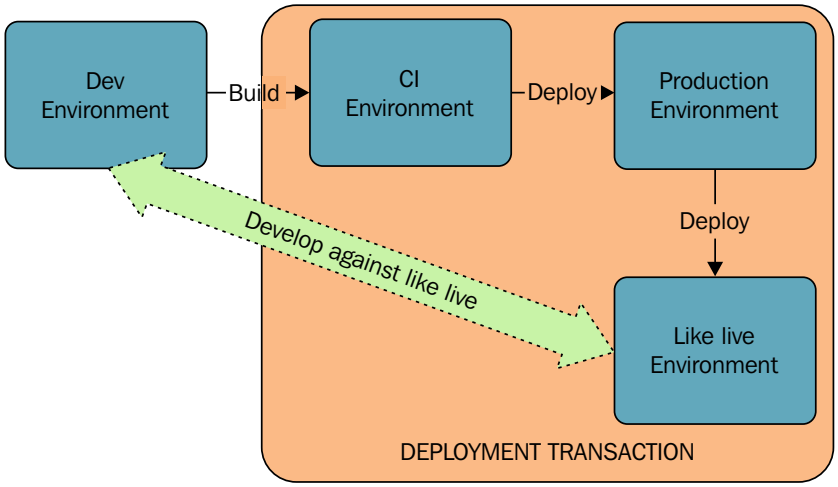
Admitting defeat, curling up in a fetal position, and lying in the corner whimpering is also not an option. As with any change, things go wrong, so review the situation, review your options, and move forward. Once you have a way to move around or even through the problem, communicate this. Ensure you're candid about what the problem is and what is being done to overcome it. This will show others how to react and deal with change—leading by example if you will.

You might be concerned that admitting failures might give the laggards more ammunition to derail the adoption; however, their *win* will be short lived once the innovators and followers have found a solution. Hold fast, stand your ground, and have faith.

Okay, so this is all very happy-clappy **positive mental attitude** (**PMA**) and might be seen by some of a cynical nature as management hot air and platitudes; however, this approach does and will work. Let's look at another real-world example.

As a simple way to manage dependencies and ensure that only one change went through to the production system at any one point in time, ACME systems implemented something they called a *deployment transaction* (see *Chapter 5*, *Approaches, Tools, and Techniques*). This worked well for a while, but things started to slow down and impact their ability to deliver. After some investigation and much debate, it turned out that the main source of the problem was that there was no sure way of determining which change would be completed before another, and there was no way to try out different scenarios in terms of integration. Simply put, if changes within asset A had some dependency on changes within asset B, then asset B needed to go live first to allow for full integration testing. However, if asset A was ready first, it would have to sit and wait—sometimes for days or weeks. The newly implemented deployment transaction itself was starting to hinder CD.

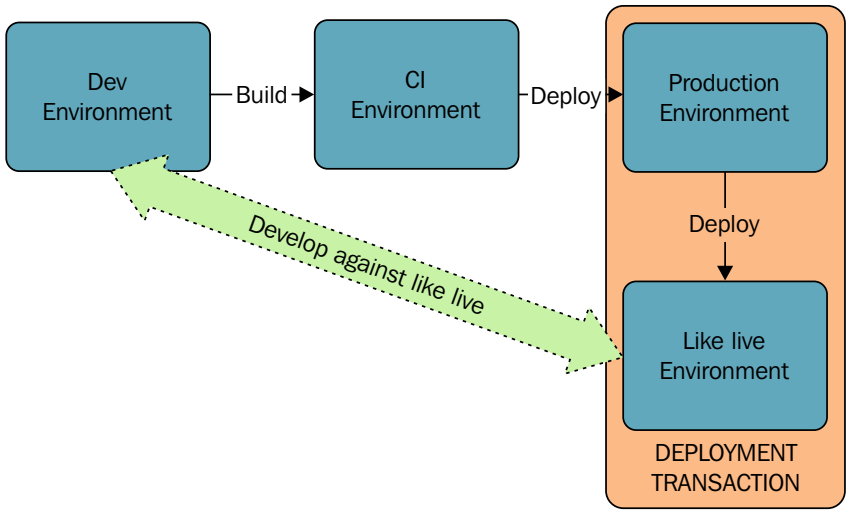
The following diagram details the deployment transaction as originally implemented:



The deployment transaction version 1.0

Everyone had agreed that the *deployment transaction* worked well and provided a working alternative to dependency hell. When used in anger, however, it started to cause real and painful problems. Even if features could be switched off through feature flags, there was no way to fully test integration without having everything deployed to production and the *like live* environment. This had not been a problem previously, as the speed of releases had been very slow and assets had been clumped together. ACME systems now had the ability to deploy to production very quickly and now had a new problem: which order to deploy? Many discussions took place and complicated tooling options were looked at, but in the end, the solution was quite simple: move the boundary of the deployment transaction and allow for full integration testing before the assets went to production. It was then down to the various engineering teams to collaborate and agree in which order things should be deployed.

The following diagram depicts the revised deployment-transaction boundary:



The deployment transaction version 2.0

So, ACME had a potential showstopper that could have completely derailed their CD and DevOps implementation. The problem became very visible, and many questions were asked. The followers started to doubt the innovators, and the laggards became extremely vocal. With some good old-fashioned collaboration and open and honest discussions, the issue was quickly and relatively easily overcome.

Again, open and honest communication and courageous dialogue is the key. If you keep reviewing and listening to what people are saying, you have a much better opportunity to see potential hurdles before they completely block your progress.

Another thing that might scupper your implementation and erode trust is inconsistent results.

Processes that are not repeatable

There is a tendency for those of a technical nature to automate everything they touch; automating the build of engineer's workstations, automated building of software, automated switching on of the coffee machine when the office lights come on, and so on. This is nothing new, and there is nothing wrong with this approach as long as the process is repeatable and provides consistent results each time. If the results are inconsistent, others will be reluctant to use the automation you spent many hours, days, or weeks pulling together.

When it comes to CD and DevOps, the same approach should apply—especially when you're looking at tooling. You need to trust the results that you are getting time and time again.

Some believe that internal tooling and labor-saving solutions or processes that aren't out in the hostile customer world don't have to be of production quality, as they're only going to be used by people within the business. This is 100 percent wrong. Internal users are as important as external ones.

Let's look at a very simple example; if you're a software engineer, you will use an **integrated development environment** (**IDE**) to write code, and you will use a compiler to generate the binary to deploy. If you're a **database administrator** (**DBA**), you'll use a SQL admin program to manage your databases and write SQL scripts. You will expect these tools to work 100 percent of the time and produce consistent and repeatable results; you open a source file, and the IDE opens it for editing; and you execute some SQL, and the SQL admin tool runs it on the server. If your tools keep crashing or produces unexpected results, you will be a tad upset (putting it politely) and will no doubt refrain from using said tools again. This might drive you insane.

> "*Insanity: doing the same thing over and over again and expecting different results.*"
>
> – *Albert Einstein*

The same goes for the tools (technical and nontechnical) you build and/or implement for your CD and DevOps adoption. You need to be confident that when you perform the same actions over and over again, you will get the same results. As your confidence grows, so does your trust in the tool/process, and you then start taking it for granted and use it without a second thought. Consequently, you will also trust the fact that if the results are different, then something has gone wrong and needs addressing.

We have already covered the potential hurdles you'll encounter in terms of corporate guidelines, red tape, and standards. Just think what fun you will have convincing the gate keepers that CD and DevOps is not risky when you can't provide consistent results for repeatable tasks. Okay, maybe fun is not the correct word; maybe pain is a better one.

Another advantage of consistent, repeatable results comes into play when looking at metrics. If you can trust the fact that to deploy the same asset to the same server takes the same amount of time each time you deploy it, you can start to spot problems (for example, if it starts taking longer to deploy, then there might be an infrastructure issue, or you might have introduced a bug within the latest version of the CD tools).

All in all, it might sound boring and not very innovative, but with consistent and repeatable results, you can stop worrying about the mundane and move your attention to the problems that need solving, such as the very real requirement to recruit new people into a transforming or transformed business.

Recruitment

This might not, on the face of it, seem like a big problem, but as the organization's output increases, the efficiency grows, and the business starts to be recognized as one that can deliver quality products quickly (and it will), then growth and expansion might well become a high priority—which is a great problem to have. You might also have lost a few laggards along the way who have decided they don't like this new and improved way of working and have moved on.

You, therefore, need to find individuals who will work in your *new way*, believe in your approach, exhibit the behaviors you have worked so hard to instill and embed throughout the organization, and, possibly, bring new skills and experience to the team. This is not an easy task, and it will take some time. Simply adding *experience in CD and DevOps* to a job spec will not produce the results you want. Although CD and DevOps are becoming prevalent approaches, they are still relatively new, and there's not that many people out there with the specific skills or experience you need. There is a growing market of recruiters who specialize in finding "DevOps engineers", but if truth be told, hardly any of them actually know what this really means—apart from adding 20 percent to the recruitment fee and salary expectations, of course.

You will need to embark on more knowledge sharing with those involved in your recruitment process to ensure that they fully understand what and who you're looking for. You might need to do this number of times until this sinks in, so be forewarned.

This might also be a good time to reach out to the CD and DevOps community and let them know you're hiring. You never know how a brief chat during the coffee break of your local DevOpsDays conference (`http://devopsdays.org/`) with a renowned engineer might pan out.

Getting potential candidates is going to be challenging. The next challenge is going to be how you filter out the good from the not so good once you have them sitting face to face in front of you. Hiring experienced and skilled engineers can be hard work normally, but when it comes to CD and DevOps, *how* they do things becomes as important (if not more important) than *what* they can do.

You'll need people who not only have proven technical experience, but also don't see the barrier between development and operations; who understand the importance of collaboration, accountability and trust; and who understand what you're actually talking about when you discuss CD and DevOps. Watch out for those who have simply read a book on the subject just before the interview—unless it's this one, of course.

You'll need to structure the interview in such a way as to tease out these intangible qualities. One example and a very simple interview question that I find works well is:

As a software engineer, how do you feel if your code was running in the production environment being used by millions of customers 30 minutes after you commit it to source control?

The question is worded specifically to get an honest emotional response, the key word here being *feel*. You will be surprised by the responses to this; for some, it simply stops them in their tracks, some will be shocked at such a thing and think you're mad to suggest it, and some will think it through and realize that although they have never considered it, they quite like the idea. If, however, the response is, *30 minutes? That's far too slow*, you might be onto a winner.

Take your time and ensure that you pick the right people. You need potential innovators and followers more than you need laggards.

Summary

As you go through the journey of implementing and adopting CD and DevOps, you will hit some bumps in the road. If you take this on board from the outset and recognize these bumps as things that are surmountable, you will be able to deal with them and continue to drive forward. As we have covered throughout this chapter, some hurdles are obvious and others not so. What they normally have in common is people or individuals who are much more difficult to analyze and debug than software or tools. It might be frustrating, but if you take your time and approach each hurdle carefully, you will reap the rewards and ultimately be successful.

Talking of success, let's now move on to the measurement of success and why it is (also) so important.

7
Vital Measurements

Over the previous chapters, we have looked at what tools and techniques you will need to successfully adopt CD and DevOps, and we highlighted some potential hurdles to overcome. With this information in hand, you should be in a good shape to succeed.

We'll now look at the important but sometimes overlooked—or simply dismissed—area of monitoring and measuring. This, on the face of it, might be seen as something that is only useful to the management types and won't add value to your CD and DevOps implementation and adoption; however, being able to understand and demonstrate progress will definitely add value to you and everyone else who is on the CD and DevOps journey. We're not just talking about simple project-management graphs and PowerPoint fodder; what we are looking at is measuring as many aspects of the overall process as possible. This way, anyone can plainly see and understand how far you have come and how far there is left to go. To be able to do this effectively, you'll need to ensure that you address this early on, as it will be very difficult to see a comparison between *then* and *now* if you don't have data representing *then*. You'll also need to ensure that you are continuously capturing these measurements so that you can compare the state of progress at different points in time.

In this chapter, you will learn:

- How to measure the effectiveness of your engineering process(es)
- How to measure the stability of the various environments you use and rely on
- How to measure the impact your adoption of CD and DevOps is having

We'll start, as they say, at the beginning and focus initially on engineering metrics.

Measuring effective engineering best practice

This is quite a weird concept to get your head around; how can you measure effective engineering, and more than that, how can you measure best practice? It's not as strange or uncommon as you would think. There are a great number of software-based businesses around the globe using tools to capture data and measurements for things such as:

- Overall code quality
- Adherence to coding rules and standards
- Code versus comments
- Code complexity
- Code duplication
- Redundant code
- Unit test coverage
- Commit rates
- Mean time between failures
- Mean time to resolution
- Bug escape distance
- Fix bounce rate

Measuring each of these in isolation might not bring a vast amount of value; however, when pooled together, you can get a very detailed picture of how things stand. In addition, if you can continuously capture this level of detail over a period of time, you can then start to measure and report on progress. You could, for example, see whether there is an impact on code quality and complexity if you reduce code duplication or redundancy.

It all sounds very simple, and to be honest, it can be, but you need to be mindful of the fact that you will need to apply some time, effort, and rigor to ensure that you gain the most value. There will also be a degree of trial and error and tweaking as you go—more inspecting and adapting—so you need to ensure that you factor this in. Not only will these sort of measurements help your engineering team(s), but they will also help with building trust across the wider business. For example, you'll be able to provide open, honest, and truthful metrics in relation to the quality of your software, which, in turn, will reinforce the trust they have in the team(s) building and looking after the platform.

One thing to seriously consider before you look at measuring things such as software code metrics is how the engineers themselves will feel about this. What Devina is thinking might be a typical reaction:

A typical reaction to this approach

Some engineers will become guarded or defensive, and see it as questioning their skill and craftsmanship in relation to creating quality code. You need to be careful that you don't get barriers put up between you and the engineering teams. You should *sell* these tools as a positive benefit for the engineers. For example, they have a way to prove how good their code actually is; they can use the tools to inspect areas of over complexity or areas of code that are more at risk of containing bugs; they can highlight redundant code and remove it from the codebase; they can visually see hard dependencies, which can help when looking at componentization; and so on.

If you have vocal dissenters, then get them actively involved in the setting up and configuration of the tools (for example, they could set the threshold at which code versus comments is set or what level of code coverage is acceptable).

If nothing more, you need to ensure that you have the innovators and followers from the engineering community brought in. To add some clarity, let's look at a few items from the preceding list—which, by the way, is not exhaustive—in a little more detail and examine why they are potentially important to your CD and DevOps adoption. Let's start with quality.

Simple quality metrics

There are a few items on the preceding list that are quite simple yet very powerful in terms of quality. The ones that are pertinent to CD and DevOps are **Mean time between failures** (**MTBF**), **Mean time to resolution** (**MTTR**), and bug escape distance, which are explained as follows:

- **MTBF**: This will help you measure how often problems (or failures) are found by end users—the longer the time between failures, the greater the stability and quality of the overall platform
- **MTTR**: This will help you measure the time taken between an issue being found and being fixed
- **Bug escape distance**: This will help you measure when an issue is found

I won't go into much more detail here, but it suffices to say that measuring these types of data will give you a good indication of progress. For example, you would expect MTBF to go up and MTTR to go down over time if CD and DevOps are working well for you. If they don't, then there's something you need to look into.

Let's now look at some other items from the list.

Code complexity

Having complex code is sometimes necessary, especially when you're looking at extremely optimized code where resources are limited and/or there is a real-time UI—basically, where every millisecond counts. When you have something like an online store, login page, or a finance module, having overly complex code can do more harm than good. Some engineers believe they are *special* because they can write complex code; however, complexity for complexity's sake is really just showing off.

Overly complex code can cause lots of general problems—especially when trying to debug or when you're trying to extend it to cater for additional use cases—which can directly impact the speed at which you can implement even the smallest change. The premise of CD is to deliver small incremental changes. If your code is too complex to allow for this, you are going to have issues down the line.

I would recommend that you put some time aside to look into this complex (pun intended) subject in more detail before you dive into implementing any process or tooling. You really need to understand what the underlying principles are and the science behind them; otherwise, this will become messy and confused. Some of the science is explained in *Appendix D*, *Vital Measurements Expanded*.

The next thing you could consider is code coverage.

Unit test coverage

Incorporating **unit tests** within the software-development process is a recognized best practice. There is lots of information available on this subject; however, simply put, it allows you to exercise code paths and logic at a much more granular and lower level during the early stages of development; this, in turn, can help spot and eradicate bugs very early on. If a small chunk of code has a good coverage, then you will be more confident that you can ship that code frequently with minimal risk—the CD approach of little and often. As you become more reliant on unit tests to spot problems, it's always a good idea to get some indication of how widespread the use is—hence the need to analyze coverage. From this, you can start to map out the areas of risk when it comes to shipping code quickly (for example, if your login page is frequently changed and has a high level of coverage, the risk of shipping this frequently becomes less).

It should be pointed out that the metric being measured is broadly based on the percentage of the codebase that is covered by tests; therefore, if the business gets hung up on this value, they might consider a low value to equate to a major risk. This isn't necessarily so, especially when you're just starting out implementing unit tests against an existing codebase that had no coverage. You should set the context and ensure that everyone looking at the data understands what it means. Maybe, you can explain to them that a handful of tests (or even one) is much better and less of a risk than none.

Let's now look at the effectiveness of measuring the frequency of commits.

Commit rates

Regular commits to source control is something that should be widely encouraged and should be deeply embedded within your ways of working. Having source code sat on people's workstations or laptops for prolonged periods of time is very risky and can sometimes lead to duplication of effort or, worse still, might block the progress of other engineers.

There might be a fear that if engineers commit too frequently, the chances of bugs being created increases, especially when you think there's an outside risk that unfinished code could be incorporated into the main code branch. This fear is a fallacy, as no engineer would seriously consider doing such a thing—why would they? The real risk is due to the fact that the longer the period of time between commits, the more code there is to be merged; this, in turn, can cause greater problems, delays, and potential bugs.

The CD approach is based on delivering changes little and often. This should not be restricted to software binaries; delivering small incremental chunks of source code *little and often* is also a good practice. If you're able to measure this, you can start seeing who is playing ball and who isn't. One word of warning: don't use this data to reward or punish engineers, as this can promote the wrong kinds of behaviors.

Next, we'll look at the thorny issue of code violations and adherence to rules.

Adherence to coding rules and standards

You may already have coding standards within your software development teams and/or try to adhere to an externally documented and recognized best practice. Being able to analyze your codebase to see which parts do and which parts don't adhere to the standards is extremely useful as it again helps highlight areas of potential risk. There are a good number of tools available to help you do this, some of which are listed in *Appendix A*, *Some Useful Information*.

This type of analysis will take some setting up as it is normally based on a set of predefined rules and thresholds (for example, info, minor, major, critical, and blocker), and you'll need to work with the engineering teams to agree and set these up within your tooling.

This all sounds like hard work—on top of all the other hard work—so is it actually worth it? Yes it is!

Where to start and why bother?

As stated earlier, there are many things that you can and should be measuring, analyzing, and producing metrics for, and there are many tools that can help you do this. You just need to work out what is most important and start from there. The work and effort needed to set up the tools required should be seen as a great opportunity to bring into play some of the good behaviors you want to embed: collaboration, open and honest dialogue, trust, and so on.

As noted earlier, it is better to implement these types of tools early in your CD and DevOps evolution so that you can start to track progress from the get-go. Needless to say, it is not going to be a pretty sight to begin with, and there no doubt be questions around the validity of doing this when it doesn't directly drive the adoption forward—in fact, things might look pretty awful, especially early on.

It might not directly affect the adoption, but it offers some worthwhile additions, which are explained here:

- Having additional data to prove the quality of the software will, in turn, build trust that code can be shipped quickly and safely
- There is a good chance that having a very concise view of the overall codebase will help with the reengineering to componentize the platform
- If the engineers have more confidence in the codebase, they can focus on new feature development without concerns about opening a can of worms every time they make a change

One thing you should also consider is including this kind of measurement and tooling within your CI process. For example, you could include *run code quality analysis* as a step within CI jobs so that the software that doesn't pass the *test* doesn't get shipped. If you think about it, you'll not only be measuring software quality, you'll also have a discreet quality gate to ensure that the code is as it should be.

We'll now move our focus from measuring the act of creating software and look at the importance of measuring what happens when it's built.

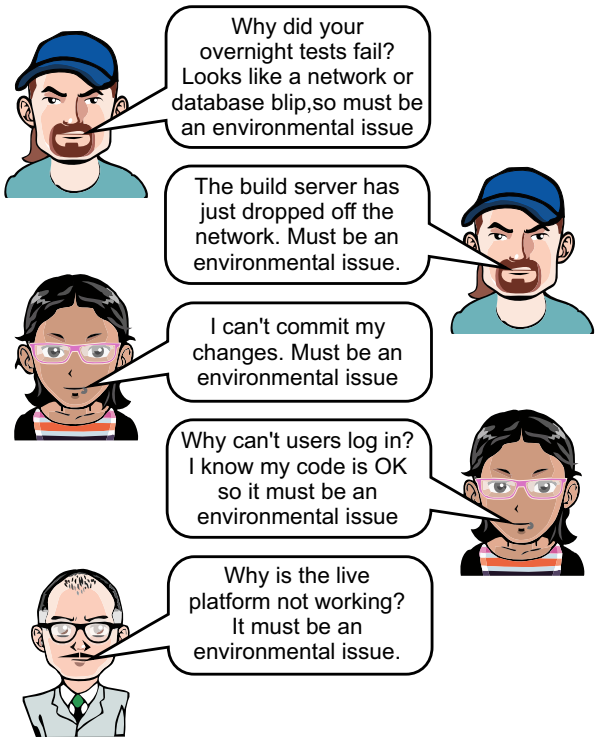
Measuring the real world

Analyzing and measuring your code and engineering expertise is one thing; however, for CD and DevOps to work, you also need to keep an eye on the overall platform, the running software, and the progress of CD and DevOps effectiveness. Let's start with environments.

Measuring the stability of the environments

As mentioned earlier, it might be that you have a number of different environments that are used for different purposes throughout the product-delivery process. As your release cycle speeds up, your reliance on these various environments will grow—if you're working in a two-to-three-month release cycle, having an issue within one of the environments for half a day or so will not have a vast impact on your release, whereas if you're releasing 10 times per day, a half-a-day downtime is a major impact.

There seems to be a universal vocabulary throughout the IT industry related to this, and the term *environmental issue* crops up time and time again, as we can see here:



The universal *environmental issue* discussion

We've all heard this, and some of us are just as guilty of saying such things ourselves. All in all, it's not very helpful and can be counterproductive in the long run, especially where building good working relationships across the Dev and Ops divide is concerned, as the implication is that the infrastructure (which is looked after by the operations side) is at fault even though there's no concrete proof.

To overcome this attitude and instill some good behaviors, we need to do something quite simple:

- Prove beyond a shadow of a doubt that the software platform is working as expected, and, therefore, any issues encountered must be based on problems within the infrastructure

Or:

- Prove beyond a shadow of a doubt that the infrastructure is working as expected, and, therefore, any issues encountered must be based on problems within the software

When I said *quite simple*, I actually meant *not very simple*. Let's look at the options we have.

Incorporating automated tests

We've looked at the merits of using automated tests to help prove the quality of each software component as it is being released, but what if you were to group these tests together and run them continuously against a given environment? This way, you would end up with a vast majority of the platform being tested over and over again—continuously in fact. If you were to capture the results of these tests, you can quickly and easily see how healthy the environment is, or, more precisely, you could see if the software is behaving as expected. If tests start failing, we can look at what has changed since that last successful run and try to pinpoint the root cause.

There are, of course, many caveats to this:

- You'll need a good coverage of tests to build a high level of confidence
- You might have different tests written in different ways using different technologies, which do not play well together
- Some tests could conflict with each other, especially if they rely on certain predetermined sets of test data being available
- The tests themselves might not be bullet proof and might not show issues, especially when they have mocking or stubbing included
- Some of your tests might *flap*, which is to say they are inconsistent and for some reason or another fail every now and again
- It could take many hours to run all of the tests end to end (on the assumption that you are running these sequentially)

Assuming that you are happy to live with the caveats or you have resources available to bolster up the tests so that they can be run as a group continuously and consistently, you will end up with a solution that will give you a higher level of confidence in the software platform. Therefore, you should be able to spot instability issues within a given environment with relative ease—sort of.

Combining automated tests and system monitoring

Realistically, just running tests will only give you half the story. To get a truer picture, you could combine your automated test results with the outputs of your monitoring solution (as covered in *Chapter 5*, *Approaches, Tools, and Techniques*). Combining the two will give you a more holistic view of the stability—or not, as the case may be—of the environment as a whole. More importantly, should problems occur, you will have a better chance at pinpointing the root cause(s).

OK, so I've made this sound quite simple, and to be honest, the overall objective is simple; the implementation might be somewhat more difficult. As ever, there are many tools available that will allow you do to this, but again, time and effort is required to get them implemented and set up correctly. You should see this as yet another DevOps collaboration opportunity.

There is, however, another caveat that we should add to the previously mentioned list:

- You might have major issues trying to run some of your automated tests in the production environment

Unless your operations team is happy with test data being generated and torn down within the production database many times per hour/day and they are happy with the extra load that will generate and the possible security implications, this approach might well be restricted to non-production environments. This might be enough to begin with, but if you want a truly rounded picture, you need to look at another complementary approach to gain some more in-depth real-time metrics.

Real-time monitoring of the software itself

Combining automated tests and system monitoring will give you useful data but will realistically only prove two things: the platform is up, and the tests pass. It does not give you an in-depth understanding of how your software platform is behaving or, more importantly, how it is behaving in the production environment being used by many millions of real-world users. To achieve this, you need to go to the next level.

Consider how a Formula One car is developed. We have a test driver sitting in the cockpit who is generating input to make the car do something; their foot is on the accelerator, making the car move forward, and they are steering the car to make it go around corners. You have a fleet of technicians and engineers observing how fast the car goes, and they can observe how the car functions (that is, the car goes faster when the accelerator is pressed and goes around a corner when the steering wheel is turned). This is all well and good, but what is more valuable to the technicians and the engineers is the in-depth metrics and data generated by the myriad of sensors and electronic gubbins deep within the car itself.

This approach can be applied to a software platform as well. You need data and metrics from deep within the bowels of the platform to fully understand what is going on; no amount of testing and observation of the results will give you this. This is not a new concept; it has been around for many years. Just look at any operating system; there are many ways to delve into the depths and pull out useful and meaningful metrics and data. Why not simply apply this concept to software components? In some respects, this is already built in; look at the various log files that your software platform generates (for example, HTTP logs, error logs, and so on), so you have a head start; if only you could harvest this data and make use of it.

There are a number of tools available that allow you to trawl through such outputs and compile them into useful and meaningful reports and graphs. There is a *but* here; it's very difficult to make this generated in real time, especially when there's a vast amount of data being produced, which will take time to fetch and process.

A cleaner approach would be to build something into the software itself, which can produce this kind of low-level data for you in a small, concise, and consistent format that is useful to you—if truth be told, your average HTTP log contains a vast amount of data that is of no value to you at all. I'll cover some examples in *Appendix D*, *Vital Measurements Expanded*, but simply put, this approach falls into two categories:

- Incorporate a *health-check* function within your software APIs; this will provide low-level metrics data when called periodically by a central data collection solution
- Extend your software platform to *push* low-level metrics data to a central data collection solution periodically

You will, of course, need something to act as the central data collection solution, but as ever, there are tools available if you shop around and work in a DevOps manner to choose and implement what works best for you.

Monitoring utopia

Whatever approach (or combination of approaches) you adopt, you should end up with some very rich and in-depth information. In essence, you'll much have as much data as your average Formula One technician (that being lots and lots of data). You just need to pull it all together into a coherent and easy-to-understand form. This challenge is another one to encourage DevOps behaviors, as the sort of data you want to capture/present is best fleshed out and agreed between the engineers on both sides.

If you're unsure whether you should measure a specific part of the platform or the infrastructure but feel it might be useful, measure it anyway. You never know whether this data will come in handy later down the line. The rule of thumb is if it moves, monitor it; if it doesn't move, monitor it just in case

Ultimately, what you want to be able to do is ensure that the entire environment (infrastructure and software platform) is healthy. This way, if someone says *it must be an environmental issue*, they might actually be correct.

If we pull all of this together, we can now expand on the preceding list:

- Prove beyond a shadow of a doubt that the software platform is working as expected, and, therefore, any issues encountered must be based on problems within the infrastructure

  Or:

- Prove beyond a shadow of a doubt that the infrastructure is working as expected, and, therefore, any issues encountered must be based on problems within the software

  Or:

- Agree that problems can occur for whatever reason and that the root cause(s) should be identified and addressed in a collaborative DevOps way

We'll now move on from the technical side of measuring and look at the business-focused view.

Effectiveness of CD and DevOps

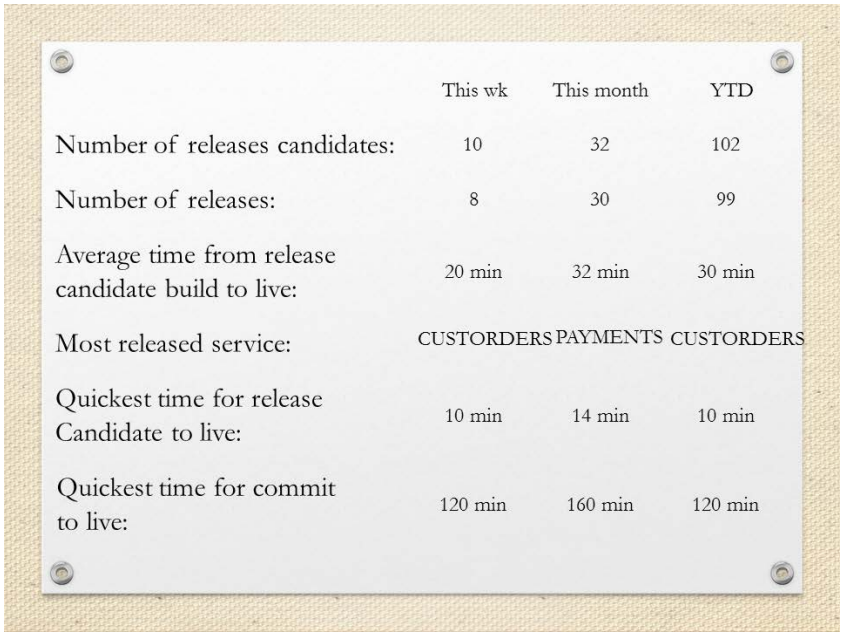
Implementing CD and DevOps is not cheap. There's quite a lot of effort required, which directly translates into cost. Every business likes to see the return on investment, so there is no reason why you should not provide this sort of information and data. For the majority of this chapter, we've been focusing on the more in-depth technical side of measuring progress and success. This is very valuable to technical-minded individuals, but your average middle manager might not get the subtleties of what it means, and to be honest, you can't really blame them. Seeing a huge amount of data and charts that contain information such as **Transactions per second** (**TPS**) counts or response times for a given software component or how many commits were made is not awe inspiring to your average *suit*. What they like is top-level summary information and data, which represents progress and success.

As far as CD and DevOps is concerned, the main factors that are important are improvements in efficiency and throughput, as these translate directly into how quickly products can be delivered to the market and how quickly the business can start realizing the value. This is what it's all about. CD and DevOps is the catalyst to allow for this to be realized, so why not show this?

With any luck, you will have (or plan to have) some tooling to facilitate and orchestrate the CD process. What you should also have built into this tooling is metrics; the sort of metrics that you should be capturing are:

- A count of the number of deployments completed
- The time taken to take a release candidate to production
- The time taken from commit to the working software being in production
- A count of the release candidates that have been built
- A league table of software components that are released
- A list of the unique software components going through the CD pipeline

You can then take this data and summarize it for all to see—it must be simple, and it must be easy to understand. An example of the sort of information you could display on screens around the office could be something like the one shown in the following screenshot:



	This wk	This month	YTD
Number of releases candidates:	10	32	102
Number of releases:	8	30	99
Average time from release candidate build to live:	20 min	32 min	30 min
Most released service:	CUSTORDERS	PAYMENTS	CUSTORDERS
Quickest time for release Candidate to live:	10 min	14 min	10 min
Quickest time for commit to live:	120 min	160 min	120 min

An example page summarizing the effectiveness of the CD process

This kind of information is extremely effective, and if it's visible and easily accessible, it also opens up discussions around how well things are progressing, what areas still need some work and optimization, and so on.

What would also be valuable, especially to management types, is financial data and information, such as the cost of each release in terms of resource and so on. If you have this data available to you, then including it will not only be useful for the management, but it could also help provide focus for the engineering teams, as they will start to understand how much these things cost.

Access to this data and information should not be restricted and should be highly visible so that everyone can see the progress being made and, more importantly, see how far they are away from the original goal.

We've looked at the effectiveness; let's now look at the real-world impact.

Impact of CD and DevOps

Implementing CD and DevOps will have an impact on your ways of working and business as a whole. This is a fact. What would be good is to understand what this impact actually is. You might already capturing and reporting against things such as business **key performance indicators** (**KPI**) (number of active users, revenue, page visits, and so on), so why not add these into the overarching metrics and measurements? If CD and DevOps is having a positive impact on customer retention, then wouldn't it be nice for everyone to see this.

At a basic level, you want to ensure that you are going in the right direction.

Before we move away from measuring and monitoring, let's look at something that, on the face of it, does seem strange: measuring your DevOps culture.

Measuring your culture

I know what you're thinking; measuring software, environments, and processes is hard enough, but how can you measure something as intangible as culture? To be honest, there are no easy answers, and it really depends on what you feel is of most value. For example, you might feel having developers working with system operators 20 percent of their time is a good indication that DevOps is working and is healthy, or the fact that live issues are resolved by developers and the operations team is a good sign.

Capturing this information can also be tricky; however, it needn't be overly complex. What you really need to know is how people feel things are progressing and if they perceive things are progressing in the correct way.

The far simplest way to capture this is to ask as many people as you can. Of course, you'll want to capture some meaningful data points—simply having a graph with the words *it's going OK* doesn't really give you much. You could look at using periodical interviews or questionnaires that capture data such as:

- Do you feel there is an effective level of collaboration between engineers (Dev and Ops)?
- How willing are engineers (Dev and Ops) to collaborate to solve production issues?
- Do you feel blame is still predominant when issues occur?
- Do you feel operations engineers are involved early enough in feature development?
- Are there enough opportunities for engineers (Dev and Ops) to improve their ways of working?
- Do you feel you have the tools, skills, and environment to effectively do your job?
- Do you feel that CD and DevOps is having a positive impact on our business?

There might be other example questions that you can think up; however, don't overdo it and bombard people—KISS (see *Chapter 3*, *Plan of Attack*). If you can use questions that allow for answers in a scale form (for example 1 being strongly agree, 2 being agree, 3 being disagree, and 4 being strongly disagree), you'll be able to get a clearer picture, which you can then compare over time.

Again, if you pool this data with your technical data, this might provide some insights you were not expecting. For example, maybe, you implemented a new process that has reduced the escaped defects by 10 percent, but releases per day have dropped by 5 percent, and the majority of the engineering team is unhappy. In such a case, you might have a problem with the process itself or rather the acceptance of it at grass roots.

Summary

Throughout this chapter, you learned that capturing data and measurements is important, as this gives you a clear indication of whether things are working and progressing in the way you planned and hoped. Whether you're interested in the gains in software quality over time, reduction in bugs, performance of your software platform, or number of *environmental issues* in the past quarter, you need data. Lots of data. Complementing this with business-focused and real-world data will only add value and provide you with more insight into how things are going.

You are striving to encourage openness and honesty throughout the organization (see *Chapter 4*, *Culture and Behaviors*); therefore, sharing all of the metrics and data you collect during your CD and DevOps implementation will provide a high degree of transparency. At the end of the day, every part of any business turns into data, metrics, and graphs (financial figures, head count, public opinion of your product, and so on), so why should the product-delivery process be any different?

The sooner you start to capture this data, the sooner you can *inspect* and *adapt*. You need to extend your mantra from monitor, monitor, and then monitor some more to monitor and measure continuously and consistently.

Let's now move from measuring everything that can and should be measured to see how things look once your CD and DevOps adoption has matured.

8
Are We There Yet?

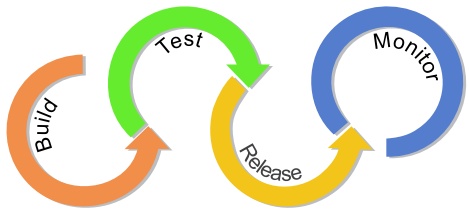
Up until this point, we have been on a journey, from surfacing the issues that caused the business pains through defining the goal and vision to remove them, addressing cultural and technical impediments, adopting the much-needed tools and techniques, and overcoming hurdles, to measuring success.

Let's presume that you are actively implementing CD and DevOps within your organization and, in fact, have been doing so for some time. The business has started to see the benefits and reap the rewards in terms of the ability to deliver quality features to the market sooner. On the face of it, you're almost done, but—and it's a very important but—this is not the end.

The journey you have all been on has been a long one, and just like the 5-year-old who has been sat in the back of the car on the long road trip to grandma's house, you will now have people within your organization repeatedly saying things such as *are we there yet?*, *how much longer?*, and *I need to pee*—okay, maybe not so much of the last one, but I think you get the point. It is now time to pause for a moment and take stock of where you are.

Reflect on where you are now

Yes, you have come a long way, yes things are going much more smoothly, yes the organization is working more closely together, yes the Dev and Ops divide is less of a chasm and more of a small crack in the ground, and yes you have almost completed what you set out to do. You have reduced the process of delivering software from something complex and cumbersome to something as simple as this:



A nice simple process for delivering software

The problems you originally set out to address revolved around the waste within the process of delivering software and, more specifically, the waste that comes from large infrequent releases. Adopting CD and DevOps has helped you overcome these problems. As a result of this, you will now start to hear comments such as *we can deploy quickly, so we must have implemented CD* or *our developers and operations people are working closely together, so we must have implemented DevOps*.

Some would suggest that once you start to hear this, it must mean that you have indeed completed what you set out to do. In some respects, this is true; however, in reality, this is far from the truth.

What these comments do illustrate is the fact that the major issues highlighted at the beginning of the journey have now started to become dim and distant memories. The business has grown to accept CD and DevOps as *the way we do things around here* and has at last started to grasp their meaning—which is good. However, you're not quite done yet. As you did at the beginning of the journey, it is again time to *inspect* and ascertain what problems are important *now*. To explain this, we have to go off on a bit of a tangent.

Streaming

Let's compare your software-release process to a river (I did say it was a bit of a tangent):

- At the very beginning, many small streams flowed downhill and converged into a river. This river flowed along, but the progress was impeded by a series of locks and a massive man-made dam.
- The river then backed up and started to form a reservoir.
- Every few months, the sluice gates were opened, and the water flowed freely, but this was normally a short-lived and frantic rush.
- As you identified and started to remove the man-made obstacles, the flow started to become more even, but it was still hindered by some very large boulders further downstream.
- You then set about systematically removing these boulders one by one, which again increased the flow; this, in turn, started to become consistent, predictable, and manageable.
- As a consequence of removing the obstacles to increase the flow, the water level starts to drop and small pebbles start to appear and create eddies, which restrict the flow to a small degree, but not enough to halt it.
- The flow goes on increasing, the water level goes on decreasing, and it soon becomes obvious that the pebbles were actually the tips of yet more boulders hidden up until this point in the depths of the river.

The flow of software resembles the flow of a river

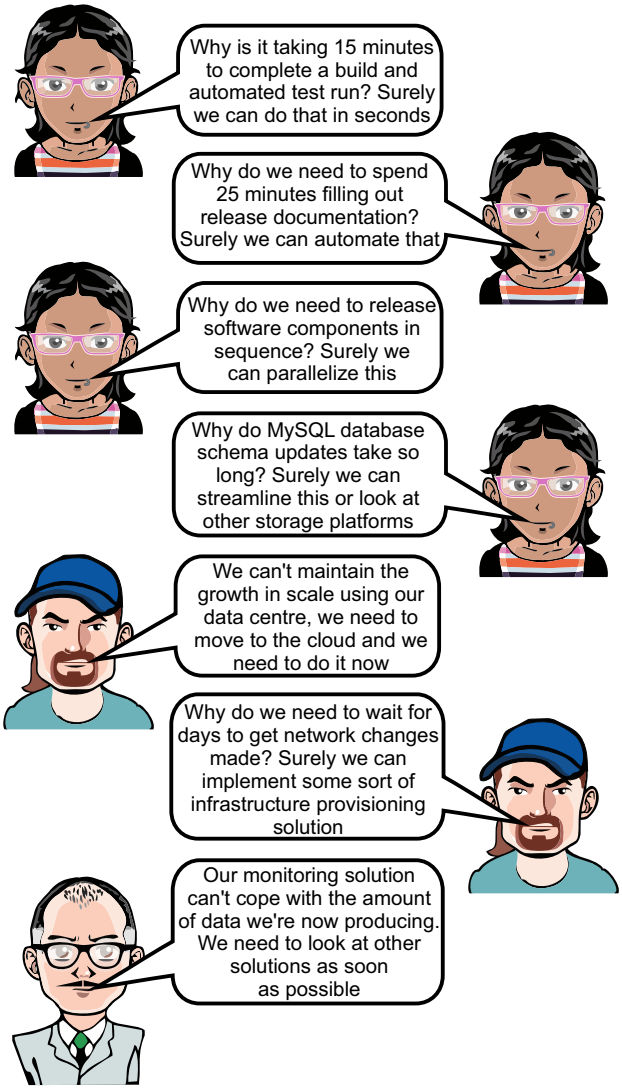
So, what's this got to do with your adoption of CD and DevOps? Quite a lot if you think about it:

- Before you started, you had many streams of work, all converging into one big and complicated release—these were the streams into the river that backed up into the reservoir.
- At the beginning of your journey, you had a pretty good idea of what the major issues and problems were. These were pretty obvious to all and were causing the most pain—these were the locks and dams.
- You removed these obstacles, and the flow started to be more consistent, but it was being hindered by the boulders—these are the lack of engineering best practice, bad culture and behaviors, lack of an open and honest environment, and so on.
- You systematically addressed and removed each of the boulders and started to get a good consistent flow, but new unforeseen issues start to pop up and impede your progress—these are the pebbles that turn out to be more boulders under the waterline.

Your original goal and vision was focused on the major issues highlighted during the *inspect* stage (the manmade locks and dams)—the things you *knew* were problems when you started out. As you systematically worked to address these, the overall process began to flow freely, and you started to see some positive and interesting results. As things progressed, hurdles (the boulders) that were not as obvious or important became more visible and a cause for concern. You then focused your efforts on removing these, which, in turn, improved the overall process. As more improvements are made, more boulders appear. All of a sudden, you have more work to do; this is something you had no way of foreseeing when you set out.

A victim of your own success

Due to the fact that things are now flowing more rapidly and smoothly, even the smallest of problems can start to become a new major issue. These problems can be relatively simple things such as:



In the space of a few months, the vast majority of the team members originally working within the constraints of big release cycles—which took many weeks or months to pull together, test, and push into the production environment—have all but forgotten the bad old, dark old days and are now finding new things to worry and grumble about. This is nothing unusual; it happens within every project, be it a major business change project or a relatively simple software-development project. It's nothing unusual, but if you think about it, it is a positive problem to have.

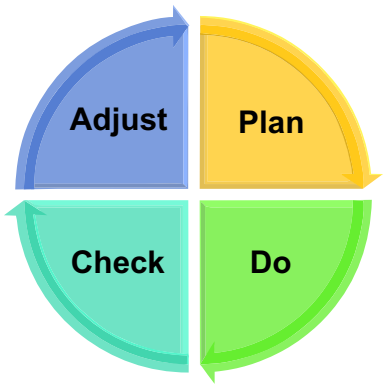
The teams were severely restricted and unable to truly innovate, experiment, or flex their engineering muscles due to the complexity and constraints of the big release process. They no longer have to worry about the process of releasing software, as this has become an everyday background noise that just happens over and over again without the need for much effort—mainly due to the excellent work you have all done.

The seemingly small problems that are now being raised would have been, in the dark days, simple annoyances, which would have been dismissed as low priority. They were pebbles. Now, they are something real, boulder shaped, and they need to be addressed; otherwise, there is a risk that things will slow down, and the days will again become darker.

Does the fact that new problems have surfaced mean that your original goal has not been met and you have failed? No it doesn't. It just means that the landscape has changed. Does this mean you need to change the goal and create a new plan? Not necessarily. What you now need is some PDCA.

[P]lan, [D]o, [C]heck, [A]djust

There are a number of variations of this acronym; however, the most widely used one is **Plan**, **Do**, **Check**, and **Adjust**. You might also find PDCA being referred to as the *Deming circle* or the *Shewhart cycle*. Whatever definition you prefer, the idea behind the PDCA approach is quite simple; it is a framework and approach that you can use for continuous and iterative improvement. The following diagram should help explain this:



The iterative PDCA process

Simply put, this approach is an expansion of the *inspect* and *adapt* approaches that have been mentioned many times previously—although, if truth be told, it's been around for much longer. The concept is pretty easy to grasp and follow and can be applied to almost every aspect of your CD and DevOps adoption. Let's look at an example:

- Plan: You realize that your current process to deliver software is broken and decide that you need to find out why, by running workshops to map out the entire process.
- Do: You run the workshops and capture input and data from across the business.
- Check: You analyze the outputs to ascertain if the data provided gives you an insight into where the pain points are within your process.
- Adjust: You highlight some areas of waste and agree on corrective actions.
- Plan: You set a goal and pull together a plan of attack to address the major pain points.
- Do: You execute against this plan.
- Check: You review the progress against the goal.
- Adjust: You make tweaks to the approach as more information and unforeseen hurdles are unearthed.
- Plan: You realign the plan to ensure that the goal is still achievable, given the new information you have gathered.
- Do ….. I think you can fill the rest in yourself.

As with most of the tools and techniques covered in this book, using the PDCA approach over any other is your call; however, it is a well-proven and well-recognized framework to use—especially when you're looking at implementing something that is as wide reaching and business changing as CD and DevOps—so, I would suggest you don't simply dismiss it out of hand.

One major advantage of PDCA is that it has the luxury of being simple to grasp and understand at all levels of the business, and it is also highly adaptable—for example, this book has been developed using the self-same approach. Do some research and make your own mind up, but before you take action, you should take a step back and take stock of where *you* are and what *you* need to do next.

Exit stage left

The business has gotten used to the changes you have all spent many long hours, days, and months implementing, and it is now experiencing new issues. The question is who should address these new found challenges? The answer is quite simple—not you.

You have helped embed the new collaborative ways of working, helped bridge the gap between Dev and Ops, helped implement new tools and optimized processes, drank lots of coffee, and had little sleep. It's now time for those you have been helping to step up.

Way back in *Chapter 2*, *No Pain, No Gain*, we looked at how to identify the problems and issues the business faced. We called this the elephant in the room. You learned how to use retrospection and other tools to look back and plan forward and how open, honest, and courageous dialogue helped to find the correct path. Now, think of the boulders in the water as a new type of elephant, and you're in exactly the same situation as you were previously. There is, however, one major and very important difference: the business now has the tools and capabilities to identify the elephant-shaped boulders very quickly and now has the tools, knowledge, confidence, experience, and expertise to remove them quickly and painlessly on their own.

As you near your original goal, your swan song is to help others help themselves. It was fun while it lasted, but all good things must come to an end. You would have reached or be very close to reaching your original goal, so now is a good time to consider your exit strategy. This isn't to say that you should not be involved at all; it just means that to fully encourage the fledging ways of working, you need to be the responsible parent and let the kids grow up and learn by their own actions.

Your focus should now change from delivering to assisting and guiding the continuation of delivery. Those who were driving the adoption of CD and DevOps—yourself included—should now start encouraging those who mostly benefitted to step into the light and take responsibility for their own boulders. It's a bit of a shift change but shouldn't be too much of a challenge—especially after all you've been through. One major parental role you can now play is to ensure that complacency doesn't set in.

Rest on your laurels (not)

So, you've done a lot, progressed further, and the business and those working within it are all the better for it. This is a positive and good thing that you and all those involved should be very proud of. However, this is no reason to rest on your laurels; it might be tempting, but now is not the time to simply sit back and admire your handy work.

You have helped the business evolve, but you have to be very mindful of the fact that the business can start to devolve just as easily if complacency sets in. As with any far-reaching project or business change, if the frantic rate of change simply stops, things start to stagnate and old ingrained habits start to resurface. The laggards might start to become noisy again, and the followers might start to listen to them.

You will have actively and notably shifted your position from *doer* to *facilitator* and *influencer*. You also need to be visible and be there to help and assist where needed. Just like a good parent, you have set up a safe environment for growth and self-discovery, and, therefore, you should only need a light touch, a bit of guiding here, some advice there, and the odd prod in the right direction.

When compared to what you have achieved, this might seem simple, but it can be much harder at times; you're used to being actively involved in driving others and doing stuff yourself and now have to help and watch others doing stuff. It's sometimes harder but just as rewarding. You have now taken the next step in your personal evolution, and as such, you are in a good position to look beyond the initial goal to see if there are opportunities to assist in a wider capacity.

Summary

Adopting CD and DevOps is a long and hard journey. If you think it's not, you are deluded. You'll circumvent elephant-filled rivers and other unforeseen challengers as you near the journey's end. Parental guidance is needed to steer the business in the right direction, while you plan how to step out of the limelight and make room for those who have benefited from the achievements you have collectively made. New problems will emerge and threaten the adoption progress; however, the business is wiser and should now have the tools, maturity, and experience to cope. Keeping an eye on things is worthwhile; however, you have bigger and better things to focus on.

9
The Future is Bright

Throughout this book, we have, on the whole, been focused on traditional software delivery within a typical and traditional web/server software-based business. Yes, there are some young, trendy, and innovative software businesses out there that have the agility and opportunity to be creative in the way they deliver software. However, the vast majority of businesses that deliver software on a day-to-day basis are not so lucky—the intention might be there, but the will to act might be lacking. Hence, the focus is on the traditional.

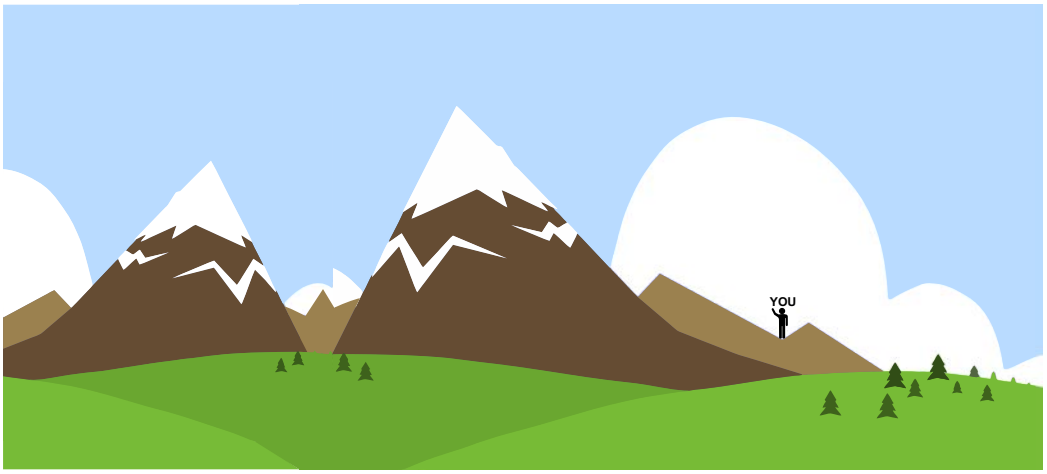
There's a strong possibility that you yourself work within one of these traditional businesses. Having followed the advice provided in this book and successfully adopted CD and DevOps, there's a very good chance that you would have caught up with the whippersnappers, and your business is able to be just as agile and creative in how it delivers software, maybe even more so.

At the tail end of *Chapter 8*, *Are We There Yet?*, we turned our focus onto you and how you could take your newfound knowledge and experience forward, beyond the initial goal of embedding the CD and DevOps ways of working within your organization. Let's look at what this could actually mean.

Expanding your horizon

Let's presume that you have been instrumental in the successful adoption of CD and DevOps and have, on the whole, delivered what you set out to do. Things are working well, even better than you envisaged. The business is all grown up, can tie its own shoe laces, and doesn't need you to hold its hand anymore—well, not quite.

Take a moment to consider where most of the individuals within the business were at the beginning of the journey and where they are now. What you will most probably find is that the vast majority are now at the same point that you were when you started out—they are just starting to realize that there is another way and that it is a better way. Now, look at how far you have come in comparison; you are so far ahead that you are not much more than a small figure in the distance.



Other's perception of you

Regardless of your role at the beginning of the journey, be that a developer, a system admin, a manager, or something else, your role has now changed. Like it or not, you have become the holder of knowledge and experience. You are the CD and DevOps subject-matter expert. You know your stuff.

You have travelled far, the landscape has changed quite dramatically from where you started, and you have new hills to climb—these are the new opportunities that the business is now ready to look at. Maybe these were challenges that the business could not overcome earlier; maybe they simply didn't know these opportunities existed, but with newfound knowledge, they are keen to try new things; maybe your **chief technology officer** (**CTO**) has been chatting with his young and trendy counterparts at the golf club. Whatever the reason, now is the time to apply *your stuff* to these new challenges and opportunities. What follows are some examples of what these could be.

Reactive performance and load testing

The more observant of you might have noticed that there is a little mention of performance or load testing throughout the book. This is intentional as, to my mind, attempting this activity without the close collaboration, tooling, and techniques that come from adopting CD and DevOps is a fool's errand. Yes, there are many established and traditional approaches, but these normally amount to shoehorning something into the process just before you want to ship your code—which might well result in the code not shipping due to the fact that performance issues were found at the last minute. I would also hazard a guess and say performance / load testing was highlighted as a major burden or even an area of waste during the *inspect* stage. It needn't be and shouldn't be the case.

Once you have adopted CD and DevOps, the act of performance / load testing can become relatively simple and straightforward. You just need to change the way you approach it.

Let's assume that you have implemented extensive monitoring of your overall platform from which you can observe in great detail what is going on under the covers. From this, you can glean an idea of how things should look during normal day-to-day operation. With this data, you should then be able to safely run controlled experiments and observe the results in terms of overall platform performance. For example, you could run an experiment to incrementally apply additional load to the platform while it's being used. As the load increases, you start to see where the pain points are—a heat map of sorts. As both Dev and Ops are working closely together, observing the platform as a whole, they should be able to work out where the problems are by comparing normal day-to-day stats to those generated under load.

If issues are pinpointed, they could even apply patches in real time using the CD tooling while the load is still in place—giving instant feedback. Alternatively, they might witness an overall slowdown of the platform, but the monitoring solution doesn't highlight anything specific. This could mean that there is a gap in the overall monitoring coverage.

All in all, trying to run performance or load testing without extensive monitoring in place and/or a high degree of collaboration between the Dev and Ops teams will not provide the results you expect or need. This is not an obvious benefit of adopting CD and DevOps, but it is a very powerful and compelling benefit, as is reducing complexity.

Reducing feature flag complexity

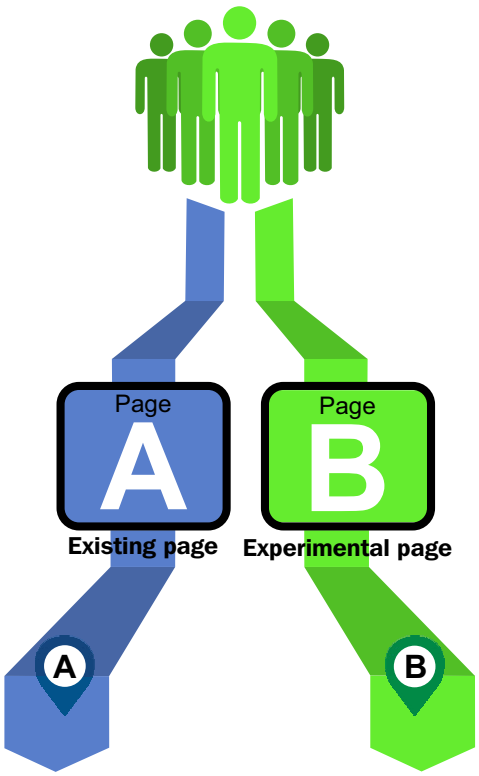
There are many established approaches to allow for different use cases or user flows to be switched on and off in real time, but most revolve around some sort of feature flag or configuration setting within the platform. Although this is a viable approach, it does add something to the code base, which can, over time, become a massive headache—complexity. It also adds complexity to the overall testing—especially if you start to chain the feature flags together (for example, feature C is *on* if feature A is *on*, but feature B is *off*).

Having adopted CD and DevOps, you will be shipping code with ease, and you'll have the Dev and Ops team working as one. Therefore, it would be far simpler to consider using the CD approach to enable and disable features or functionality. In other words, to enable the feature, you just ship the code with it enabled—no messing around with flags, settings, or chaining. OK, so this is an overly simplistic view, but with CD and DevOps, you can start looking at these sorts of problems in new and innovative ways. The advantages might not be immediately obvious, but reducing complexity, if nothing else, will save you time, effort, and the business money. Something that can drive the necessity of feature flags is A/B testing.

Easing A/B testing

Another major benefit to come out of adopting CD and DevOps is the relative ease by which you can implement an A/B-testing approach. I won't go into too much depth regarding this subject—there are plenty of books and on-line resources you could read. However, the top level is this: A/B testing gives you the ability to run different use-cases in parallel and examine the results to see which approach (A or B) worked best.

Maybe you want to see what the impact would be if you introduced a new design or web-page layout. If you can, in some way, force some users down path A and the rest down path B, you can then monitor the user behavior to see which worked best. You can also run A/B experiments covertly. For example, if you have a new recommendation service that you want to try out, you could again force some user traffic to this and see how it works compared to the incumbent service. The possibilities are endless.



A simplistic example of A/B testing

You don't *need* CD or DevOps to implement A/B testing; however, CD does give you the ability to ship code quickly—for example, you want to implement the code to split traffic to A or B across all servers in minutes so that all users start using the same code at the same time. You also have Dev and Ops closely working together, monitoring everything that is going on. If gaps are found in the data used to analyze the results, you have the ability to address this with relative ease. You also have the option to roll everything back if things take an unexpected turn.

Without CD and DevOps, you would need to plan this kind of activity very closely in advance and hope nothing is missing or amiss when you implement it. You will also have to hope that the world hasn't moved on too much between planning and execution and that the use-case you are trying to test is still relevant. Something else that is time critical is security patching.

Security patching and saving your bacon

It seems that every day, the tech press includes a report about the latest business that has been hacked or suffered a **distributed denial-of-service** (**DDOS**) attack. Let's apply this scenario to a traditional software business that has not adopted CD or DevOps. Now think of the answers you would get if you asked the following questions:

- How quickly do you think your average traditional software business would apply a patch to overcome the problem?
- How calm do you think their operations team feels with their CEO, VP of PR, and SVP of operations, all breathing down their collective necks?
- How confident are the Ops team that hastily applying an O/S patch that should have been applied months ago will not impact the software platform?
- How happy do you think the development team will be when the SVP of engineering tells them that they can't go home until they have sorted out a fix to overcome the issues introduced by hurriedly applying an O/S patch?
- How much market value do you think is wiped off a listed company when the news gets out?

It doesn't take a PhD to guess the answers to these questions. Now, imagine the same situation for a business that has adopted CD and DevOps. The answers to the preceding questions would be something like this:

- As quickly as they can normally release—minutes or hours at the most.
- Perfectly calm, and, to be honest, the senior management wouldn't know anything about it until they've been informed that an issue had been found during routine monitoring and was in the process of being addressed.
- Very confident as they can collaborate with the development team to ensure that there are no impacts and/or work on a plan to address the impacts in parallel.
- They won't have to.
- If the message delivered is "We found an issue and have already applied a fix. The impact was minimal and we can assure our customers that their data is perfectly safe", the news isn't very newsworthy, and the markets might not even care. In fact, they might even see it as good news and want to invest more in the business (OK, I can't quantify this, but it is plausible).

As you can see, adopting CD and DevOps can provide some major *bacon-saving* benefits. That isn't to say that you couldn't achieve the same results without CD and DevOps, but having the ability to deliver quickly and having a very close working relationship across the Dev and Ops teams will make it much easier to spot and fix live issues before anything breaks. To reinforce this approach, some people actually try to actively break their live platform on purpose.

Order out of chaos monkey

It doesn't matter how much care and attention you apply to your platform; something will always go wrong when you least expect it. A server will fail, a process might start looping, a network switch decides it doesn't want to be a network switch anymore, a pipe bursts in the office above the server room, or someone decides to hack you because you're a nice big target. As the saying goes, you need to expect the unexpected.

Most businesses will have some sort of business contingency plan in place to cater for the unexpected, but there's a strong possibility that they don't try to force the issue on purpose—they just hope nothing bad will ever happen, and if it does, they hope and pray that they'll be ready and the plan works.

What if you had some tools that could safely initiate a failure at will to see what happens and, more importantly, where the weak spots in your platform are? This is exactly what some bright spark did, and this has been widely adopted as the *chaos monkey* approach. There are a few variations, but what it boils down to is this: a tool that you can run within your closely monitored environment to try and break it.

The tools currently available are very much focused on cloud-based installations, but the approach could be applied to other environment setups.

If you tried to attempt this without a strongly embedded CD and DevOps culture, you would end up in a complete mess—to be honest, I doubt if you would be even be allowed to try it in the first place. With close collaboration, in-depth monitoring, and trust-based relationships, attempting to break the platform to observe what happens is relatively (but not totally) risk free.

There is one caveat to this approach; you need to be confident that your platform has been designed and built to gracefully allow for failure. You should avoid committing *platformicide* in public with core dumps and HTTP 500 messages available for all to see.

One other advantage to the *chaos monkey* approach is that it's also a great way to share knowledge of how the overall platform works. Trying to break things on purpose can prove the resilience of your overall platform, but might well annoy customers; let's take a look at how we can keep your customers happy.

End user self-service

Over the course of the book, we have been focused on a unidirectional process of *pushing* software out to a given environment (including production). What if you wanted to turn this around and allow end-users to initiate the *pulling* of your software at will? It might sound strange, but there are a few legitimate scenarios where this could be required.

Maybe you have a team that would like to test out different scenarios and use cases or a SecOps team that needs to run a set of deep security scans or some DDOS scenarios. Traditionally, this would involve quite a large amount of mundane work (that's putting it very mildly) to set up a dedicated environment and get all of the software needed installed and working as it should. What if these teams could press a button and have an entire environment set up for them and they could trust that it has been set up correctly?

With CD and DevOps embedded into your ways of working, you should be able to do this. You will have tooling that can reliably provision servers, deploy software assets, and provide in-depth monitoring. You have a DevOps team who are used to collaborating and are, therefore, happy to help the teams in question. You have a culture that is open and honest so that if problems are found, they can be openly discussed and investigated without fear of blame. Finally, you have the ability to quickly fix and ship if need be. All in all, the activity becomes valuable and not in the least bit mundane or wasteful.

You can most probably think of other scenarios but the point is that with CD and DevOps embedded within your ways of working, you are able to take the load off the Ops and Dev team and at the same time have happy internal customers. This approach can also help build trust—something that again is a pretty powerful asset. Continuing the theme of making customers happy, let's look at how you can widen your horizons and apply CD and DevOps in other ways.

CD and DevOps and the mobile world

CD and DevOps are normally associated with delivering server-based solutions—that's not to say it is exclusively the case; however, this is the norm. The fact that CD and DevOps are based on enhancing culture, behaviors, and ways of working means that they needn't be constrained to this flavor of software delivery. You could, should, and can apply the same approaches to other flavors, for example, the development and delivery of mobile apps.

As mentioned earlier, CD and DevOps are based on culture, behaviors, and ways of working, therefore applying this approach to delivering mobile applications—which is a large and ever-growing industry—can work. There are a couple of caveats in terms of how delivering mobile application software differs from web-based / server-based software delivery.

They are explained here:

- Delivering software to a web platform 10 times per day seamlessly without impacting the end user is achievable—you are in full control of the infrastructure and the mechanism for releasing. Doing the same with a mobile application will have a major impact on the end user—can you imagine what will happen if you send a mobile app to end users' smart phones 10 times per day?
- There are no system operators living within the end users' smart phones / tablets; therefore, the Ops side of the DevOps partnership doesn't strictly exist

So, how do you square this circle? In reality, you don't need to. Most mobile apps are distributed via some sort of app store, which is simply a third-party binary repository—similar to what you would use when shipping binary files to your servers within your CD toolset. When the user wants to install the app, they initiate a process to pull it and install it (very similar to the self-service approach mentioned earlier). When updates are published, the user (or the operating system) simply pulls the binary and installs it.

All pretty simple stuff, so why should you apply CD and DevOps to this? Because you can, and it will add value. The work undertaken to embed collaboration, trust, and honesty within your organization can easily be applied to developing, building, testing, and shipping your mobile apps. You have implemented tools and techniques to automate the process of building, testing, shipping, and monitoring your server platform, so extending these for your mobile apps should be relatively straightforward.

Added to this is the fact that mobile apps can be written in the same technologies as you would use on a server-based website—HTML5, for example. This, in turn, means that the same code base could potentially be shipped to both server and mobile; therefore, using the same techniques, tools, and approaches will make the process seamless.

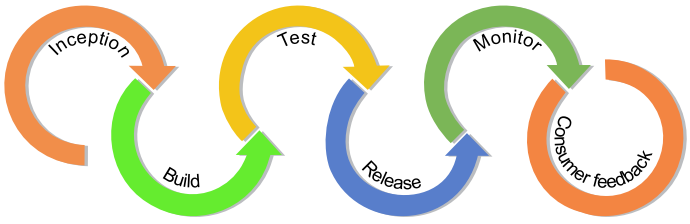
One thing you will need to look into is slightly tweaking the approach in terms of CD. There's nothing stopping you from releasing your app 10 times per day to prove that your process works or to allow for rigorous beta testing. However, you should seriously consider how often you publish this to the app store so that your end users aren't bombarded with app updates' notifications. On the whole, this is a minor issue. Another slight tweak would be to apply the CD and DevOps ways of working outside the world of traditional software delivery.

Expanding beyond software delivery

Despite what you have read, CD and DevOps need not be restricted to simply software/product delivery. The tools, processes, and best practices that come with this way of working can be extended to include other areas of the business outside of traditional IT. For example, let's presume that your product-delivery process is now optimal and efficient, but there are certain business functions that sit before and after the actual product-delivery process; these functions are starting to creak, or, maybe, they are starting to hinder the now highly-efficient product-delivery stage.

There is no reason why using the same techniques covered earlier you cannot address wider reaching business problems. You now have the experience, confidence, and respect to take something that is unwieldy and cumbersome and streamline it to work more effectively, so why not apply this further afield? For example, you could extend the overall product-delivery process to include the *inception* phase (which normally sits before the product-delivery process and is sometimes referred to as the *blue-sky* phase) and the *customer-feedback* phase (which normally sits after you have delivered the product).

The following diagram illustrates this:



Extending the product-creation process to include the pre and post stages

Doing this could provide even greater business value and allow more parts of the business to realize the huge benefits of the CD and DevOps ways of working. As I say, CD and DevOps is not just about delivering software; the way things get done, the collaboration, the open and honest environment, the trust-based relationships, and even the language used can and will help revitalize any business process. It's worth considering whether this is what you would like to do.

What about me?

The preceding are simply examples, but none will have the chance of becoming a reality without someone helping the business and steering it in the right direction. Like it or not, you will have the experience, skills, and reputation as the *go-to guy* for things related to CD and DevOps.

You now have the opportunity to start a new journey and again help the business help itself by driving forward the sort of changes that can only be realized with a mature and strong CD and DevOps culture.

If this doesn't float your boat, then, maybe, keeping up with the ever-changing and ever-growing CD and DevOps landscape is your thing. Just trying to keep up with the new ways to do things, new tools, new ideas, and new insights could take most of your time and attention. More and more businesses are realizing the huge value of having evangelists in their ranks—especially when it comes to software and product delivery.

You might well have hooked yourself into the global CD and DevOps communities, which will give you opportunity to share or present your experiences with others and, more importantly, bring others' experiences and knowledge back into your business. Maybe you could even capture this and publish it on public blogs and forums, or even get it printed in book form. Stranger things have happened.

Whatever you choose to do, you will not be bored, nor will you be able to go back to how things were. You have learned a very valuable lesson—there is a better way, and CD and DevOps is it.

What have you learned?

I keep making references to your experience, knowledge, and expertise, but until you have actually gone through the motions of adopting and implementing CD and DevOps, this will amount to what you have read. Let's take a final chance to recap what we have covered:

- CD and DevOps are not just about technical choices and tools; a vast amount of the success is built on the behaviors, culture, and environment.
- Implementing and adopting CD and DevOps is a journey that might seem long and daunting at first, but once you've taken the first step and then put one foot in front of the other, you'll hardly notice the miles passing.
- Teams who have successfully implemented CD and DevOps seldom regret it or are tempted to go back to the bad old days when releases were synonymous with working weekends and late nights—working late nights and weekends should be synonymous with innovation and wanting to create some killer app or the next world-changing technology breakthrough.
- You don't have to implement both CD and DevOps at the same time, but one complements the other. You don't have to, but you should seriously consider it.
- Where you do need to make technical choices, ensure that you implement something that enhances and complements your ways of working—never change your ways of working to fit the tooling.
- It can be big and scary, but if you start with your eyes wide open, you should be able to get through. There is a global community available that can help, assist, and give advice, so don't be afraid to reach out.
- Don't simply start implementing CD or DevOps just because it's the next new thing that everyone else is doing. You need to have a good reason to implement both/either, or you will not reap the benefits, nor truly believe in what you are doing.
- Although we have covered a vast amount, you don't have to implement everything you have read about; take the best bits that work for you and your situation, and go ahead from there—just as you would with any good *agile* methodology.
- Just because you can ship software, it doesn't mean you are done. CD and DevOps are ways of working, and the approaches within can be applied to other business areas and problems.
- Share failures and successes so that you learn, and others have the opportunity to learn from you.

Summary

This book, like all good things, has come to an end. As pointed out numerous times earlier, we've covered quite a lot in a few pages. This book is, by no means, the definitive opus for CD and DevOps; it is merely a collection of suggestions based on experience and observations.

Even if you are simply window shopping and looking at what is needed to implement and adopt CD and DevOps ways of working, you should now have a clearer idea of what you are letting yourself and your organization in for; forewarned is forearmed as they say. It's not an easy journey, but it is worth it.

So, go grab yourself a hot beverage, a notepad, and a pen; skip back to *Chapter 2*, *No Pain, No Gain*; and start mapping out why you need to implement CD and DevOps and how you are going to do it.

Go on then, stop reading, go!

Good luck!

A
Some Useful Information

Although this book provides some (hopefully) useful information, there's only so much space available. I've, therefore, compiled a list of additional sources of information that will complement this book. I've also included a list of the many subject-matter experts out there who might be able to provide further assistance and guidance as you progress along your journey. Additional resources can be found on my website `http://www.swartout.co.uk`.

What follows is, by no means, an exhaustive list, but it is a good start.

Tools

Some of the following tools are mentioned within this book, and some are considered the best of breed for CD and DevOps:

Tool	Description	Where to find more information
Jenkins	An award-winning and world-renowned open source CI tool	`http://jenkins-ci.org/`
GIT	A free and open-source distributed version-control system	`http://git-scm.com/`
GitHub	An online-hosted community solution based on GIT	`https://github.com/`
Graphite	A highly-scalable real-time graphing system that allows you to publish metric data from within your application	`http://graphite.wikidot.com/`
Tasseo	A simple-to-use Graphite dashboard	`https://github.com/obfuscurity/tasseo`

Tool	Description	Where to find more information
SonarQube	An open platform to manage code quality	`http://www.sonarqube.org/`
Ganglia	A scalable distributed-monitoring system for high-performance computing systems	`http://ganglia.sourceforge.net/`
Nagios	A powerful monitoring system that enables organizations to identify and resolve IT-infrastructure problems before they affect critical business processes	`http://www.nagios.org/`
Dbdeploy	An open source database change management tool	`http://dbdeploy.com/`
Puppet Labs	A tool to automate the creation and maintenance of IT infrastructure	`http://puppetlabs.com/`
Chef	Another tool to automate the creation and maintenance of IT infrastructure	`https://www.getchef.com/chef/`
Vagrant	A tool to build complete development environments using automation	`https://www.vagrantup.com/`
Docker	An open platform for distributed applications	`https://www.docker.com/`
Yammer	An Enterprise private social network (think of it as a corporate Facebook)	`https://www.yammer.com`
HipChat	A private group-chat and team collaboration tool	`https://www.hipchat.com/`
IRC	The granddaddy of collaboration and chat tools	`http://www.irc.org/`
Campfire	A private group-chat and team collaboration tool	`https://campfirenow.com/`
Hubot	An automated "bot" that can be set up within most chat-room systems	`https://hubot.github.com/`
Trello	An online scrum / Kanban board solution	`https://trello.com/`
AgileZen	An on-line scrum / Kanban board solution	`http://www.agilezen.com/`

People

What follows is a list of people who are actively involved in the agile and continuous delivery and DevOps communities:

- Patrick Debois is seen by many in the DevOps community as the daddy of DevOps and the founder of the DevOpsDays movement (`http://devopsdays.org/`). This relatively small get together of like-minded individuals in 2009 has grown into a global gathering. For more information on Patrick Debois, visit `http://www.jedi.be/`.
- John "Botchagalupe" Willis is a regular and well-renowned contributor to the DevOps community and has inspired many with his honest way of sharing his wisdom. For more information on him, visit `http://www.johnmwillis.com/`.
- Jez Humble is the co-author of the *Continuous Delivery* book that is used by many as the definitive reference material when investigating or implementing continuous delivery. He also actively contributes to the continuous-delivery blog (`http://continuousdelivery.com/`). For more information on him, visit `http://jezhumble.net/`.
- John Allspaw is the SVP of Operations at Etsy.com and seems to understand the value of DevOps—even though he's one of the senior management types. For more information on him, visit `http://www.kitchensoap.com/`.
- Gareth Rushgrove is a self-confessed Web geek, who seems to somehow find time to produce the DevOps weekly e-mail newsletter (`http://devopsweekly.com/`), which is full of useful and insightful information. For more information on him, visit `http://www.garethrushgrove.com/`.
- Gene Kim, co-author of *The Phoenix Project*, is the founder and former CTO of Tripwire. He is passionate about IT operations, security, and compliance, and how IT organizations successfully transform from *good to great*. For more information on him, visit `http://www.realgenekim.me/`.
- Mitchell Hashimoto is a self-confessed DevOps tools mad scientist and the creator of Vagrant, Packer, Serf, Consul, and Terraform. For more information on him, visit `http://about.me/mitchellh`.
- Steve Thair is the cofounder of DevOpsGuys and a regular speaker on Web performance. For more information on him, visit `http://www.devopsguys.com/About/Team/Steve`.
- Rachel Davies is an internationally-recognized expert in coaching teams on the effective use of agile approaches and has a wealth of knowledge when it comes to retrospective techniques and games. For more information on her, visit `http://www.agilexp.com/agile-coach-rachel-davies.php`.

- Ken Schwaber is the godfather of scrum and agile. For more information on him, visit `http://kenschwaber.wordpress.com/`.
- John Clapham is an all-round nice guy and agile/DevOps evangelist. For more information on him, visit `http://johnclapham.wordpress.com/`.
- Karl Scotland is a renowned agile coach who specializes in lean and agile techniques. For more information on him, visit `http://availagility.co.uk/`.

Recommended reading

The following books are well worth a read—even if you don't decide on some strange reason to adopt CD and/or DevOps:

Resource	Description	Link
Agile Coaching	A nice introduction on how to become a good agile coach	`https://pragprog.com/book/sdcoach/agile-coaching`
Agile Retrospectives: Making Good Teams Great	An excellent book that covers most of what you need to know to run effective retrospectives	`https://pragprog.com/book/dlret/agile-retrospectives`
Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation	The CD bible	`http://www.amazon.com/dp/0321601912?tag=contindelive-20`
The phoenix project	A unique take on DevOps adoption in fiction form, well worth a read	`http://itrevolution.com/books/phoenix-project-devops-book/`
Agile Product Management with Scrum	View scrum and agile from the product managers' point of view	`http://www.amazon.com/exec/obidos/ASIN/0321605780/mountaingoats-20`

Resource	Description	Link
The Enterprise and Scrum	This book provides some addition insight into the challenges of adopting an agile approach and ways of working	`http://www.amazon.com/exec/obidos/ASIN/0735623376/mountaingoats-20`
The Lean Startup	Real-life experiences and insights into how to transform your business, culture, and ways of working	`http://amzn.com/0307887898`
Getting Value out of Agile Retrospectives	Gives a good introduction on retrospectives and provides a good, long list of games/exercises	`https://leanpub.com/gettingvalueoutofagileretrospectives`

B

Where Am I on the Evolutionary Scale?

It's sometimes difficult to ascertain where your business sits on the CD and DevOps evolutionary scale; however, some simple questions can help you get a rough idea. For example, in relation to your business:

- Does it favor process over people?
    1. Process
    2. People
    3. We don't have any processes worth mentioning, so I suppose it's people
- Do immoveable deadlines in project plans take precedence over delivering quality solutions incrementally?
    1. Yes, meeting deadlines is the only thing that matters
    2. We have flexibility to make small changes and replan to ensure that quality doesn't suffer
    3. We do whatever is needed to keep the customer happy
- Are your projects run with fixed timescales, fixed resources, and fixed scope, or is there flexibility?
    1. Yes, and this is all agreed up front, signed off, and intricately planned
    2. No, we have flexibility in at least one of these areas
    3. We do whatever is needed to keep the customer happy

- Is failure scorned upon or used as something to learn from?
    1. Failure is failure, and there are no excuses—heads will roll
    2. We ensure that failures have a small impact and learn from our mistakes
    3. Failure means no more business, and we're all out of our jobs
- Who is on call for out-of-hours production issues?
    1. The T1 helpdesk with T2 operations support and T3 applications support teams backing them up
    2. We normally have a "point man" on call who can reach out to anyone he needs
    3. Everyone
- Are you able to ship code when it is ready, or do you have to wait for a scheduled release?
    1. The release team schedules and agrees the delivery to production via the change advisory board (CAB) and transition team, based on the agreed program plan
    2. We trust our engineers to ship code using our deployment tools when they are confident that it is ready and doesn't compromise the overall quality
    3. Our engineers normally use file transfer protocol (FTP) to transfer the code to the production servers when it's finished compiling

If you were to apply these to the ACME systems' business at certain points through their evolution, you would find that the version 1.0 business would mostly answer *3* to all questions, the version 2.0 business would mostly answer *1*, and the highly evolved version of the business would mostly answer *2*.

C
Retrospective Games

Retrospectives are normally the *inspect* part of the agile *inspect and adapt*. If you are aware of or are using scrum or some other agile methodology, then running retrospectives should be nothing new. If you have never run a retrospective before, then you would have some fun things to learn.

The remit of a retrospective is to look back over a specific period of time, project, release, or simply a business change and highlight what worked well, what didn't work well, and what improvements are needed. This process can traditionally be a bit dry, so retrospectives tend to be based on games (some people refer to these as "exercises", but I prefer the word "games"), which encourages collaboration, engagement, and injects a bit of fun.

As with any game, there are always rules to follow. Here are some example rules:

- Each session should be strictly time-boxed
- Everyone should be given a voice and a chance to actively contribute
- Everyone should be able to voice their opinion but not at the expense of others
- Whoever is facilitating the session is in charge and should control the session as such
- The session should result in tangible and realistic actions that can be taken forward as improvements

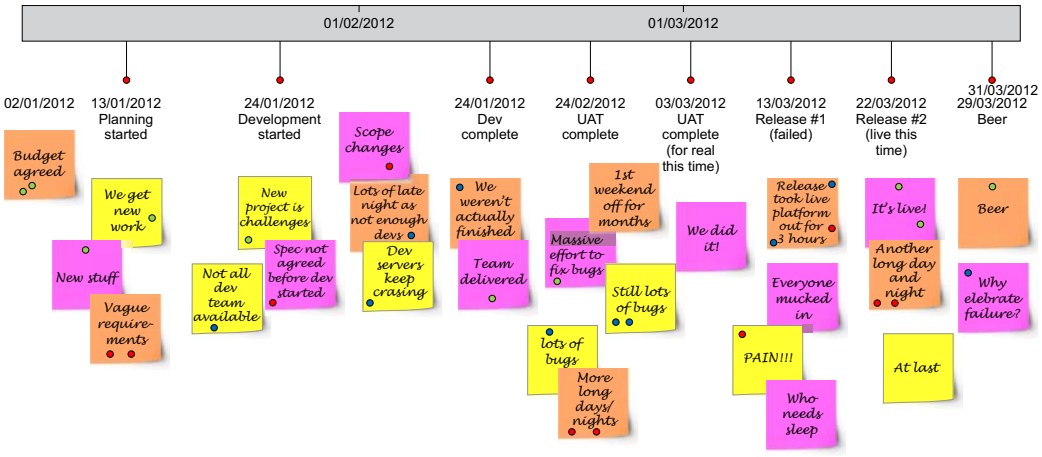
As with the value-stream mapping technique mentioned in *Chapter 2*, *No Pain, No gain*, the only tools you really need are pens, paper, a whiteboard (or simply a wall), some space, and some sticky notes.

Let me introduce you to a couple of my favorite games: timeline and *StoStaKee*. We'll start with the timeline game.

The timeline game

The timeline game, as the name suggests, revolves around setting up a timeline and getting your invited guests to review and comment on what happened during the period of time in question. There are a number of variations of this game, but in essence, it revolves around the entire team writing out sticky notes related to notable events during the period in question and indicating how the events made them feel using sticky dots (*green* stands for *glad*, *blue* for *sad*, and *red* for *mad*). From this, you have an open and honest discussion on those events that provoked the most emotions and agree on actions to take forward (for example, things to stop doing, start doing, and keep doing).

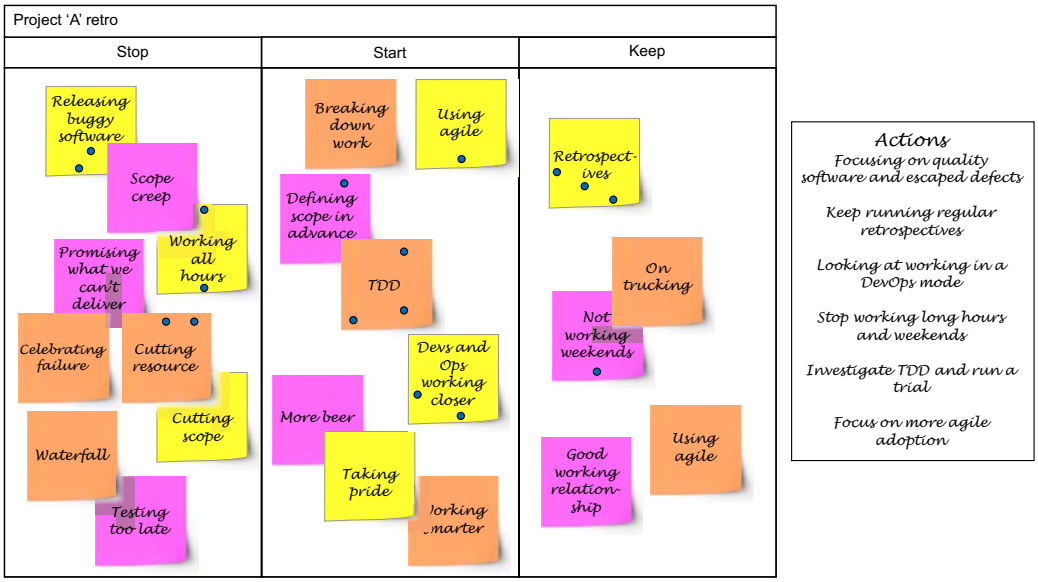
The following figure depicts a typical timeline wall:



StoStaKee

This stands for **stop**, **start**, and **keep**. Again, this is an interactive time-boxed exercise focused on past events. This time, you ask everyone to fill in sticky notes related to things they would like to stop doing, start doing, or keep doing, and add them to one of three columns (stop, start, and keep). You then get everyone to vote—again with sticky dots—on the ones they feel most strongly about. Again, you should encourage lots of open and constructive discussions to ensure that everyone understands what each note means. The end goal is a set of actions to take forward.

The following figure depicts a typical StoStaKee board:



The preceding examples are a mere subset of what is available, but both have proven time and time again to be the most effective in investigating and, more importantly, understanding the issues within a broken process.

D
Vital Measurements Expanded

Chapter 7, *Vital Measurements*, introduced you to a number of different ways of measuring certain aspects of your processes. We will now expand on some of these and look in more detail at what you could/should be measuring. We'll start by revisiting code complexity and the science behind it.

Code complexity – some science

As mentioned in *Chapter 7*, *Vital Measurements*, having complex code in some circumstances is fine and sometimes necessary; however, overly complex code can cause you lots of problems, especially when trying to debug or when you're trying to extend it to cater to additional use cases. Therefore, being able to analyze how complex a piece of code is should help.

There are a few documented and recognized ways of measuring the complexity of source code, but the one most referred to is the cyclomatic complexity metric (sometimes referred to as MCC or McCabe Cyclomatic Complexity) introduced by Thomas McCabe in the 1970s. This metric has some real-world science behind it, which can, with the correct tools, provide quantifiable measurements based on your source code. The MCC formula is calculated as follows:

$M = E - N + X$

In the preceding formula, M is the MCC metric, E is the number of edges (the code executed as a result of a decision), N is the number of nodes or decision points (conditional statements), and X is the number of exits (return statements) in the graph of the method.

Code versus comments

Including comments within your source will make it much more readable, especially in the future when someone other than the original author has to refactor or bug fix the code. Some tools will allow you to measure and analyze the ratio of code versus comments.

That said, some software engineers don't believe that comments are worthwhile and believe that if another engineer cannot read the code, then they're not worth their salt. This is one view; however, including comments within one's source should be encouraged as a good engineering practice and good manners.

One thing to look out for should you implement a code-versus-comments analysis is those individuals who get around the rules by simply including things such as the following code snippet:

```
/**
 * This is a comment because I've been told to include comments in my
 * code
 * Some sort of code analysis has been implemented and I need to
 * include comments to ensure that my code is not highlighted as poor
 * quality.
 *
 * I'm not too sure what the percentage of comments vs code is
 * required but if I include lots of this kind of thing the tool will
 * ignore my code and I can get on with my day job
 *
 * In fact this is pretty much a waste of time as whoever is reading
 * this should be looking at the code rather than reading comments.
 * If you don't understand the code then maybe you shouldn't be trying
 * to change it?!?
 */
```

This might be a bit extreme, but I'm sure if you look close enough at your codebase, you might well find similar sorts of things hidden away.

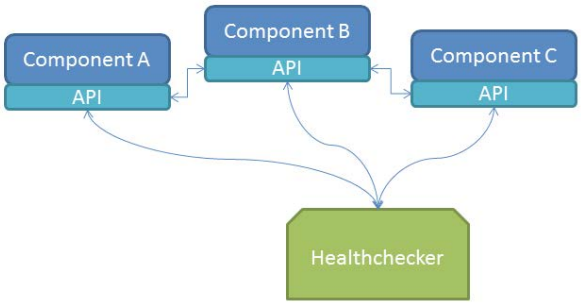
One other good reason for comments—in my experience—is for those situations when you have to take the lid off some very old code (by today's standards, very old could be a couple of years) to investigate a possible bug or simply find out what it does. If the code is based on outdated design patterns or even based on an old language standard (for example, an older version of Java or C#), it might be quite time-consuming trying to understand what the code is doing without, at least, some level of commenting.

Embedding monitoring into your software

As mentioned in *Chapter 7*, *Vital Measurements*, there are a few ways you can include and embed the generation of metrics within the software itself.

Let's assume that your software components contain APIs that are used for component-to-component communication. If you were able to extend these APIs to include some sort of a health-check functionality, you could construct a tool that simply calls each component and asks the component how it is. The component can then return various bits of data, which indicates its health. This might seem a bit convoluted, but it's not that difficult.

The following diagram gives an overview of how this might look:



A health-checker solution harvesting health-status data form software components

In this example, we have a health-checker tool that calls each component via the APIs and gets back data that can then be stored, reported, or displayed on a dashboard. The data returned can be as simple or complex as you like. What you're after is to ascertain whether each component is healthy. Let's say, for example, one element of the data returned indicated whether or not the software component could connect to the database. If this comes back as false and you notice that the system monitor looking at the free disk space on the database server is showing next to zero, you can very quickly ascertain what the problem is and rectify it.

This method of monitoring is good but relies on you having some tooling in place to call each component in turn, harvest the data, and present it to you in some readable/usable form. It's also restricted to what the APIs can return or rather how they are designed and implemented. If, for example, you wanted to extend the data collection to include something like the number of open database connections, you will need to change the APIs, redeploy all of the components, and then update the tooling to accept this new data element. This is not a huge problem, but a problem all the same. What could be a huge problem, though, is the single point of failure, which is the tooling itself. If this stops, working for whatever reason, you're again blind, as you don't have any data to look at, and, more importantly, you're not harvesting it.

There is an alternative approach that can overcome these problems. In this approach, the component itself generates the metrics you need and pushes the data to your tooling. Something like Graphite does this very well. Instead of extending the APIs, you simply implement a small amount of code; this allows you to fill up buckets of metrics data from within the software component itself and push these buckets out to the Graphite platform. Once in graphite, you can interrogate the data and produce some very interesting real-time graphs. Another advantage of graphite is the plethora of tools now available to generate and create very effective graphs, charts, and dashboards based on the Graphite data.

Index

D

E

P

R

S

T

U

V

Y

Thank you for buying

Continuous Delivery and DevOps – A Quickstart Guide

Second Edition

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

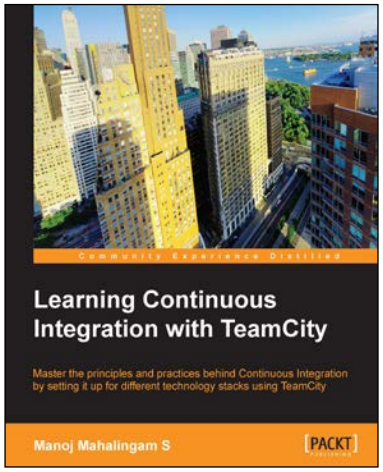
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

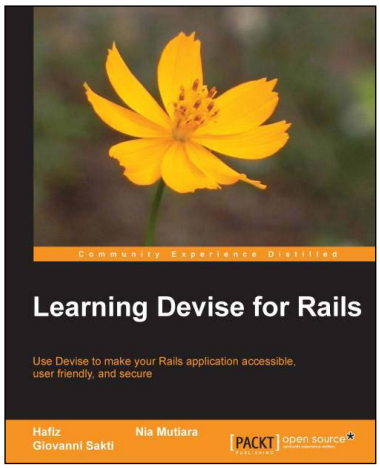


Learning Continuous Integration with TeamCity

ISBN: 978-1-84969-951-8 Paperback: 276 pages

Master the principles and practices behind Continuous Integration by setting it up for different technology stacks using TeamCity

1. Learn about the features that TeamCity brings to the table to make setting up and practicing CI easy.
2. Enable team, organization and self to start using TeamCity for CI, from scratch or from an existing setup.
3. Setup CI for Java, .NET, Ruby, Python and mobile projects using TeamCity.

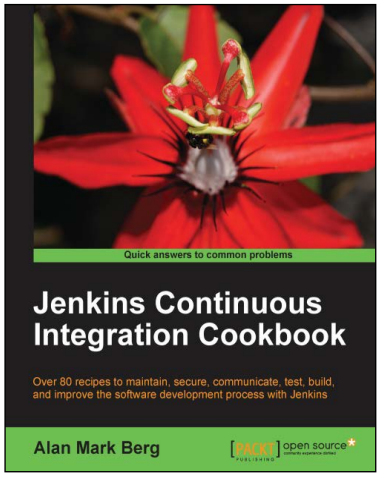


Learning Devise for Rails

ISBN: 978-1-78216-704-4 Paperback: 104 pages

Use Devise to make your Rails application accessible, user friendly, and secure

1. Use Devise to implement an e-mail-based sign-in process in a few minutes.
2. Override Devise controllers to allow username-based sign-ins, and customize default Devise HTML views to change the look and feel of the authentication system.
3. Test your authentication codes to ensure stability.

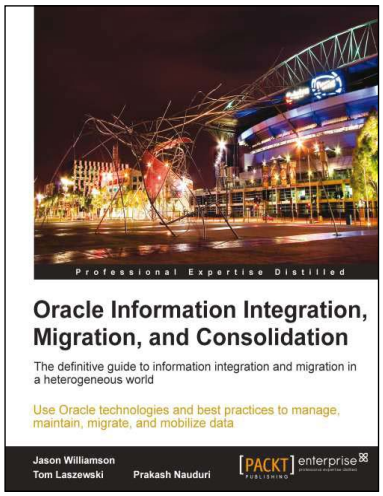


Jenkins Continuous Integration Cookbook

ISBN: 978-1-84951-740-9 Paperback: 344 pages

Over 80 recipes to maintain, secure, communicate, test, build, and improve the software development process with Jenkins

1. Explore the use of more than 40 best of breed plugins.
2. Use code quality metrics, integration testing through functional and performance testing to measure the quality of your software.
3. Get a problem-solution approach enriched with code examples for practical and easy comprehension.



Oracle Information Integration, Migration, and Consolidation

ISBN: 978-1-84968-220-6 Paperback: 332 pages

Use Oracle technologies and best practices to manage, maintain, migrate, and mobilize data

1. Learn about integration practices that many IT professionals are not familiar with.
2. Evaluate and implement numerous tools like Oracle SOA Suite and Oracle GoldenGate.
3. Get to grips with the past, present, and future of Oracle Integration practices.