Community Experience Distilled

Continuous Delivery and DevOps : A Quickstart Guide

Continuous delivery and DevOps explained

Paul Swartout

[PACKT] PUBLISHING

Continuous Delivery and DevOps: A Quickstart Guide

Continuous delivery and DevOps explained

Paul Swartout

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Continuous Delivery and DevOps: A Quickstart Guide

First published: October 2012

Production Reference: 1181012

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

ISBN 978-1-84969-368-4

www.packtpub.com

Cover Image by Abhishek Pandey (abhishek.pandey1210@gmail.com)

Credits

Author
Paul Swartout

Reviewers
Daniel Albu
Jacob Kottackal Ninan
Leonardo Risuleo
Veturi JV Subramanyeswari

Acquisition Editor
Wilson D'Souza

Commissioning Editor
Meeta Rajani

Technical Editor
Kirti Pujari

Copy Editor
Laxmi Subramanian

Project Coordinator
Michelle Quadros

Proofreader
Aaron Nash

Indexer
Hemangini Bari

Graphics
Aditi Gajjar
Valentina D'Silva

Production Coordinator
Prachali Bhiwandkar

Cover Work
Prachali Bhiwandkar

About the Author

Paul Swartout has spent over 20 years working in IT. Starting out as a Junior Developer within a small software house, Paul has filled a number of roles over the years including Software Engineer, System Administrator, Project Manager, Program Manager, Operations Manager, 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 multi-national corporates.

Paul is passionate about software and how it is delivered. Since first encountering "agile" almost a decade ago he has been committed to the adoption and implementation of agile techniques and approaches to improve efficiency and output for software development teams.

Until very recently Paul headed up the team responsible for delivering continuous delivery solutions into the Nokia Entertainment business. Paul and his team spent the best part of a year changing the default ways of working and driving the adoption of CD and DevOps as the de facto mode of delivery for Nokia Entertainment products.

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

Paul is a software development manager at Nokia and is based within the Nokia entertainment team in Bristol in the UK. The entertainment team is responsible for designing, building, and running the entertainment services and solutions for Nokia customers around the globe. These products include Nokia Music, Nokia Reading, and Nokia TV.

Acknowledgement

Firstly I would like to say a big thank you to my darling wife Jane, who has 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 lots of things—which eventually turned into this book.

Thank you to my daughter Harriet for her proof reading skills—I knew that education would come in handy one day.

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

I would also like to thank Deon Fourie—one of the best bosses out there—and my CD team for their faith and commitment to delivering the CD and DevOps vision. Without you this body of work would be rather dry and many pages lighter.

A big thank you to John Fisher for allowing me to include his transition curve within the book and to Patrick Debois for his support and never ending enthusiasm and drive to bring DevOps to the masses.

About the Reviewers

Daniel Albu is a freelance Interactive Developer who specializes in Flash Platform and HTML5 based solutions, with more than 10 years of experience in production and deployment of rich media websites, games, and applications.

Daniel has unique expertise in the development and delivery of Flash Platform-based technologies and HTML5 solutions.

Moreover, Daniel provides worldwide remote training and consultancy services around Flash Platform and HTML5 based solutions for companies and individuals alike.

Visit Daniel's website: `http://www.danielalbu.com`

Read his blog: `http://www.flashdeveloper.co`

Or follow him on Twitter: `@danielalbu`

Jacob Kottackal Ninan is a Project Manager with a Bachelor of Engineering and MBA degree. He has worked globally for companies like International Air Transport Association (IATA) and eBay.

Jacob is passionate about technology. He also takes up consulting assignments in management and business.

Jacob was a contributor to the 5th edition of the Project Management Book of Knowledge (PMBOK) for the Project Management Institute, USA.

He was also a member and Independent Consultant at Technical Working Group, World Resources Institute / WBCSD Scope 3 Emissions Protocol Development contributing to the GHG emission standards.

Jacob co- founded Kottackal Business Solutions Pvt. Ltd., which is behind the e-commerce hotel booking portal for the Indian market (`www.luxotel.in`).

He can be contacted at `jacobninan@gmail.com`

I would like to thank my father K. C. Ninan, mother Mariamma Ninan, and my wife Irina for their support.

Leonardo Risuleo is a designer and developer with several years experience in mobile, new media, and user experience (`http://www.leonardorisuleo.info/`). He's a highly dedicated professional and passionate about what he does. He started back in 2003 and during these years he worked on a variety of different mobile and embedded platforms for a number of well-known brands. Leo designs, prototypes, and develops mobile applications, games, widgets, and websites.

Apart from being a Flash Platform enthusiast Leo also contributes to the Flash and mobile community as an author and blogger, and he's co-founder of the Italian Mobile & Devices Adobe User Group. From 2008 to 2010 Leo had the honor to be the Forum Nokia Champion, a recognition and reward program for top mobile developers worldwide.

In 2010 he formally founded Small Screen Design (`http://www.smallscreendesign.com/`), a design and development studio focused on mobile design and user experience.

Veturi JV Subramanyeswari is currently working as a Solution Architect at a well-known software consulting MNC in India. Post joining this company she served a few Indian MNCs, many start ups, and R&D sectors in various roles such as Programmer, Tech Lead, Research Assistant, Architect, and so on. She has more than eight years of experience working with web technologies covering media and entertainment, publishing, healthcare, enterprise architecture, manufacturing, public sector, defense communication, gaming, and so on. She is also a well-known speaker who delivers talks on Drupal, Open Source, PHP, Women in Technology, and so on.

She has reviewed other tech books, some of which are as follows:

- *Drupal 7 Multi Sites Configuration*
- *Building Powerful and Robust Websites with Drupal 6*
- *Drupal 6 Module development*
- *PHP Team Development*
- *Drupal-6-site-blueprints*
- *Drupal 6 Attachment Views*
- *Drupal E-Commerce with Ubercart 2.x*
- *Drupal 7: First Look*
- *Twitter bootstrap*
- *Drupal SEO*

I would like to thank my family and friends who supported me in completing my reviews on time with good quality.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

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.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across 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 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 nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

For a while now, there has been a buzz around the IT industry about something called continuous delivery and DevOps. Strictly speaking that should be "some things" as continuous delivery and DevOps are actually two separate things.

- 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
- Continuous delivery is not a methodology you can use to speed up large releases which happen every few months
- DevOps is another way of working whereby developers and system operators work in harmony with little or no organizational barriers between them towards a common goal
- DevOps is not a way to get developers doing operational tasks so that you can get rid of the operations team and vice versa

Just like any other agile methodology, tool, or way of working you are not forced to use everything you read or hear concerning continuous delivery and DevOps. You simply need to understand what bits they are made up of and which of these bits would bring most value to you and your organization.

This book will provide you with some background information into these two new kids on the block and how they can help you to optimize, streamline, and improve the way you work and ultimately how you ship quality software. Included within are some tricks and tips based upon real-world experiences, which may help you reduce the time and effort needed if you were to do it alone.

What this book covers

Chapter1, *Evolution of a Software House* introduces you to ACME systems and the evolution of the business, from fledgling start-up through the growing pains following acquisition by a global corporate, to get the best of both worlds.

Chapter 2, *No Pain, No Gain* introduces some of the introspective techniques that can be used to determine what the current pain points are within the software delivery process and try to understand where they stem from.

Chapter3, *Plan of Attack* gives you some pointers into how the success of implementing continuous delivery and DevOps can be defined and success measured.

Chapter 4, *Tools and Technical Approaches* will give you some options around the tooling that can help with the implementation of continuous delivery and DevOps—right from basic engineering best practices to advanced, fully automated solutions.

Chapter 5, *Culture and Behaviors* highlights the importance of the *human* factors that must be taken into account.

Chapter 6, *Hurdles to Look Out For* will give the reader some useful tips and tricks for overcoming or avoiding the bumps in the road as they go through their evolution.

Chapter 7, *Measuring Success and Remaining Successful* focuses on the various key performance indicators and measures that can be used to monitor and communicate the relative success of continuous delivery and DevOps adoption. We also look at what to do when it's all working.

Appendix, *Some Useful Info* provides you with some more detailed information about the tools referenced within the book and some useful contacts within the global continuous delivery and DevOps community.

This appendix is only available as a free download at `http://www.packtpub.com/sites/default/files/down0loads/Some_Useful_Info.pdf`.

What you need for this book

There are many tools mentioned in this book that will help you no end. These include technical tools, such as Jenkins, GIT, Sonar, and graphite and non-technical tools and techniques, such as Scrum, Kanban, agile, and TDD.

You may have some of these (or similar) tools in place or are looking at implementing them, which will help. However, the only things 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 freelance software developer, a system administrator working within a corporate business, an IT project manager, or a CTO in a startup, 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 continuous delivery or DevOps, this book should give you an insight into what all the fuss is about. If you have already set out to adopt continuous delivery and/or DevOps, then this book may help provide some useful tips and tricks. If you know everything there is to know about both/either of the subjects, then this book will help reaffirm your choices and may provide some additional things to chew over.

All in all the target audience is quite broad; it includes anyone who wants to understand how to ship quality software regularly without the pain.

Conventions

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

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 may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through 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 on `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.

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 would 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/support`, 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.

Piracy

Piracy of copyright 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

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Evolution of a Software House

Before we get into the meat of what continuous delivery (CD) and DevOps are all about, I would like to introduce you to ACME systems. This fictional web-based software business started out—as these things sometimes do—in a back bedroom of one of the founders. The founders were visionaries with big ambitions, good ideas, dreams, and a little bit of cash.

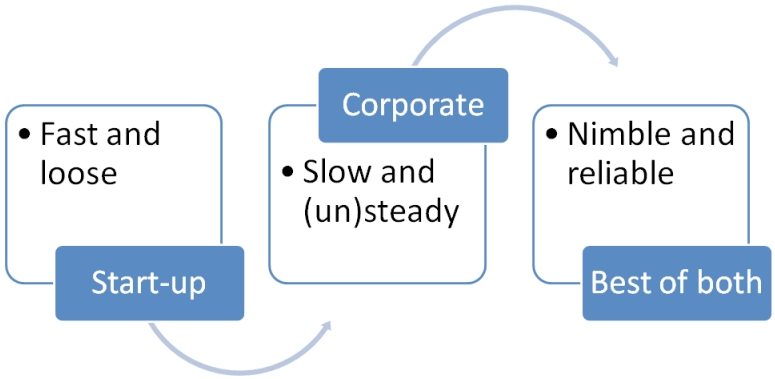
After a few years of hard work, determination, and much blood, sweat and tears, the dream of the founders was realized. The business was recognized as a leader in their field and was acquired by a multinational corporate. This acquisition brought with it the funding and resources needed to allow the business to grow and expand to become a global player. That is all well and good, but the story doesn't end there. With corporate owners comes corporate responsibilities, corporate rules, corporate bureaucracy, and corporate processes.

The ACME systems team—now embellished with the corporate owner's name and now operating under corporate governance—started to find it increasingly difficult and painful to deliver quality software. They adopted and adhered to the parent company's processes to improve quality and reduce risk but this made the simple task of delivering software, laborious and extremely complex.

They came to an evolutionary crossroad and had to make a decision—live with the corporate baggage that they had inherited, try and get back to the good old days and good old ways, which had reaped benefits previously, or try to take the best of both worlds.

It so happens that they made the right choice to implement CD and DevOps and did indeed reap the best of both worlds.

The following diagram gives an overview of the evolution of ACME systems:



Overview of ACME systems evolution

Over the next few pages, we'll go through this evolution in 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 previously mentioned, ACME systems started out in humble beginnings; the founders had the big vision and could see a gap in the market for a web-based solution. They had entrepreneurial spirit and managed to obtain backing from a few parties who injected the life blood 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, which they had seen before anyone else could.

At first, it was slow going and hard work; lots 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, long nights, long weeks, long weekends, and long months, things started to come together. The customer base started to grow and the orders started rolling in; so did the revenue. Soon the number of employees were in double figures and the founders had become directors.

This is all well and good but 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 the 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 is, getting software into the production environment as quickly as possible and delighting customers
- Speed of delivery is of the essence
- When things break, everyone swarms around to help fix the problem
- 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 has to operate to stay alive:

- Corners will be cut to hit deadlines, which will compromise software design and elegance
- Engineering best practices will be compromised to hit deadlines
- Testing will not be 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 is not used religiously

- With unrestricted access to the production environment, tweaks and changes can be made to the infrastructure
- 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

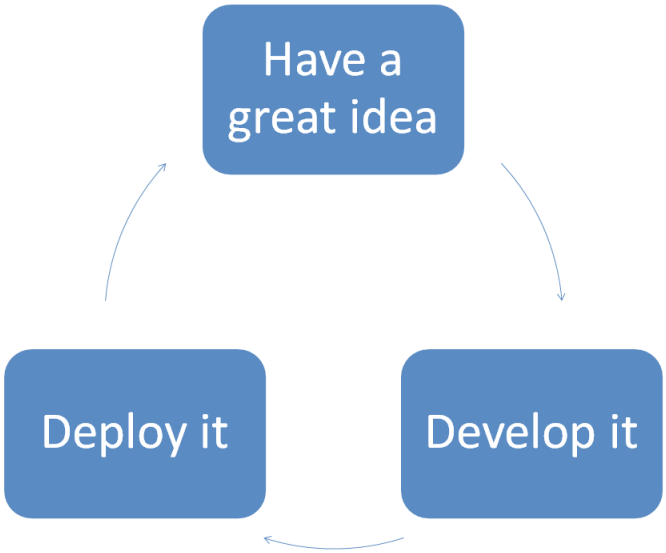
To emphasize this, let's have a look at a selection of typical conversations between the Development and Operations teams and the management team of ACME systems.

Developer/Ops	**Management**
We would like to invest some time in implementing a source control system.	Is it free? We don't have time.
We would like to invest time in developing a fully automated test suite.	Is it free? We don't have time.
We would like to invest time in implementing automated server provisioning.	Is it free? We don't have time.
The production environment has crashed.	You're not going home until it's fixed. I'll get the pizza!
This prototype is rough and ready and needs to be rewritten before we hand it over to our customers.	Don't worry, you'll get time to rewrite it.
The prototype we are now using in production keeps crashing.	You're not going home until it's fixed. I'll get the pizza!
We want to work in a test-driven development mode.	That will only slow things down and we don't have the time.
I can manually hack the production server to improve performance and stability to overcome the issues we're having.	I fully trust your judgment on this, just get it done quickly.
The manual hack I did last week has caused the disks to fill up and the production server has crashed.	You're not going home until it's fixed. I'll get the pizza!

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.



Overview of ACME 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 of the field.

ACME systems Version 2.0

The business has grown in size, numbers, and turnover. The customer base spans a number of countries and the ACME 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 their area of expertise.

So much so, that the board of ACME systems is approached by a well-known and established multinational corporation and discussions are entered into regarding an acquisition. The discussions go very well and an agreement is reached; ACME systems is acquired outright. The board members are extremely happy with this 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, re-architecting the software platform to be scalable and performant, investing in new technologies, tools, and R&D. The drier side of business—administration, program, and 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 number of months. The R&D team—as they're now called—introduce 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 platform starts to creak and is showing its age, so further investment is provided to completely re-architect and rewrite the 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 Operations team is augmented 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 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 resource available to help maintain quality, control change to the platform, and ensure the platform is stable and available 24 x 7.

Sitting above all of this, the parent company also has well-established 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 all that rosy at all.

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

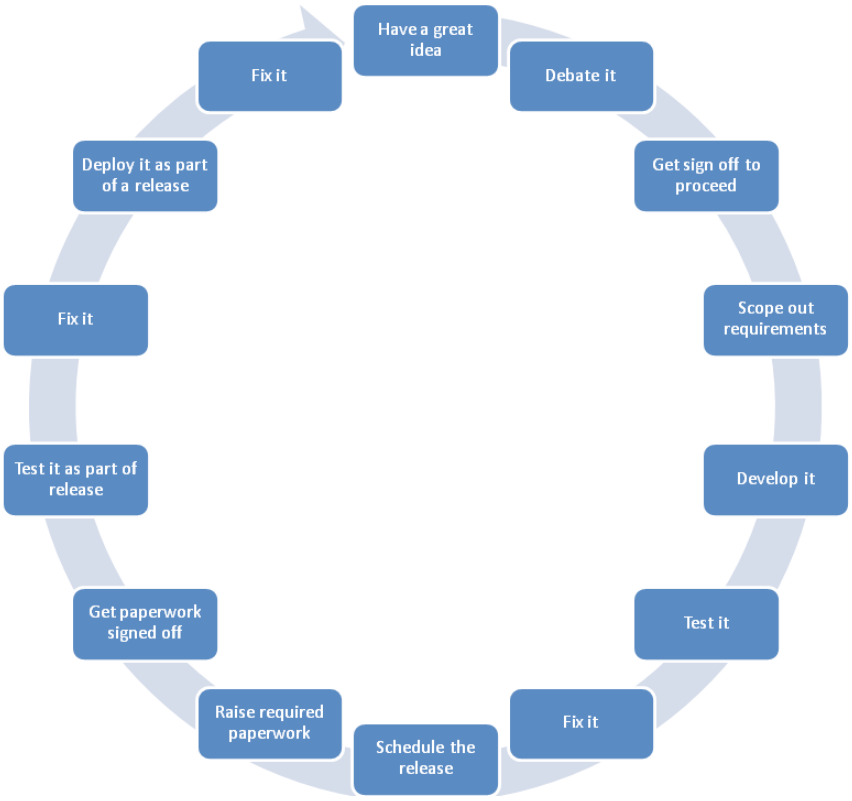
- It is getting increasingly difficult to ship software—what took days now takes weeks or even months
- Releases are getting more and more complex as the new platform grows and more integrated features are added
- Developers are now far removed from the production environment and as such are ignorant as to how the software they are writing performs, once it eventually goes live
- Focus is being lost as new people come into the organization with no prior knowledge of the platform or the business
- 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
- The software quality is starting to suffer as last minute changes and frantic bug fixes are being applied to fit into release cycles
- 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 resource is being applied to assisting releases, which is impacting 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 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 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.



Overview of ACME 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, this time between the R&D and Operations teams (who you'll remember were pretty much one and the same in the early days of ACME systems).

R&D	Operations
We need to urgently fix a performance bug and need to check some of the system configuration values for one of the production servers.	Have you obtained permission to see this information?
We have now obtained permission to see the system configuration values for one of the production servers. Can you please supply it?	Which server?
We've no idea—the one running the secure web login service.	You'll have to be more specific, we've got hundreds of servers.
We've checked through an old project plan and it's listed as DC02MM03DB16.	That sounds like a database server, it shouldn't be running code. Application servers normally have AP rather than DB.
Maybe it's DC02MM03AP16?	That sounds more like it. We've found it. What information do you need?
The heap size for the JVM.	It's 16 GB. However, the server's only got 8 GB of RAM.
That's not good, no wonder there are performance issues. Can we increase it up to 16 GB? That's the minimum space the application needs.	You'll have to raise a change ticket.
Okay—for now I'll set the deployment parameters to use 8 GB, can you update the system configuration?	That's an infrastructure change. You'll have to raise an infrastructure change ticket.
What about spinning up a couple of new servers so we can spread the burden?	That's a DC infrastructure change. You'll have to raise a DC infrastructure change ticket.
We now have all the tickets raised and signed off. We're now ready to deploy this fix.	As the heap size has changed, we need to see the results from integration, performance, and functional tests as this deployment could have an adverse impact on the production platform.
Arrggghhh! I give up!!	

Okay, so this may be a little over the top but it just goes to highlight the massive disjoin between the R&D and Operations team(s). This communication is now normally via e-mail.

As was previously stated, not everyone noticed the attrition within the organization—luckily a few brave souls did.

A few brave men and women

A small number of the ACME team can see the issues within the process as clear as day and 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 100 MPH to get software delivered at all costs.

At first, they form a small virtual team, start breaking down the issues, and try to implement and/or build tooling to ease the pain:

- Tools to automate software builds
- Automated testing suites
- Continuous integration (CI) systems
- Automated *no downtime* deployment tools
- Clones of the production environment for functional and performance testing using virtualization technologies

This goes some way to address the issues but there is still one fundamental problem that tooling cannot address—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 at the same time obtaining backing to address it.

We're now going on to the third stage of the evolution where things start to come back together and the 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 and becomes dedicated to addressing the culture and behavior and developing ways to overcome and/or remove the barriers. They are no longer simply a Development 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 CD and DevOps.

The first thing they do is to go out and simply talk with as many people as possible, across the business, who are involved in the process of getting software from conception to consumer. This not only gathers useful information but also gives the team the opportunity to evangelize and form a wider circle of like-minded individuals.

They have a vision, purpose, and they passionately believe in what needs to be done.

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

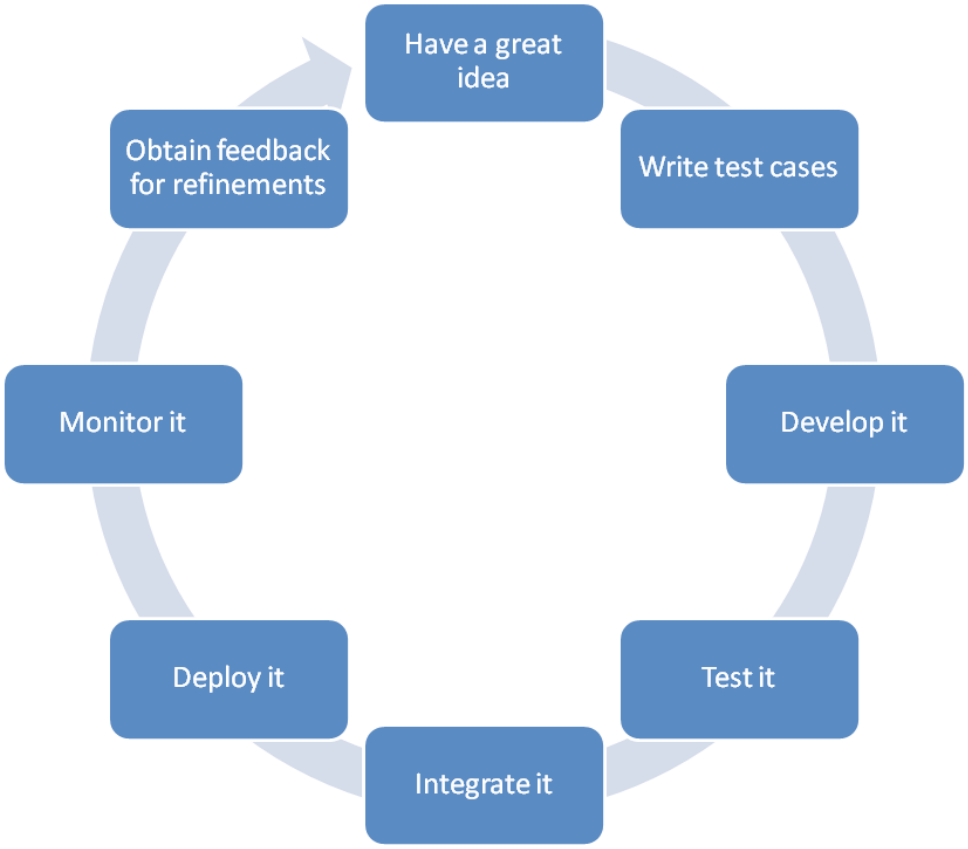
- Running various in-depth sessions to understand and map out the end to end product delivery process
- Refining and simplifying the tooling based upon continuous feedback from those using it
- Addressing the complexity of managing dependencies and order of deployment
- Engaging experts in the field of CD to independently assess the progress being made
- Arranging CD and DevOps training and ensuring that both R&D and Ops team members 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 implement other 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 nine months or so the process of building and delivering software has transformed to the extent that a code change could be built, fully tested, and deployed to the production platform in under 18 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 have a final look 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 it 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.



Overview of ACME 3.0 software delivery process

This highly efficient process has freed up valuable Development and Operations resources so that they can get back to their day jobs—developing and delivering new software features and ensuring that the production platform is healthy and customers are again delighted.

The ACME systems team has got back its mojo 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.

Summary

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

Whatever point in the evolution a software-based business sits, there are some very simple and fundamental truths that are universal:

- Developers enjoy solving problems by writing code and delivering software for consumers to use
- System operators enjoy solving problems by implementing technical solutions and running stable platforms
- Businesses who rely on software and hardware to generate revenue want both to be optimal and work in harmony

Being in a position to fully realize and accommodate all of the above will reap rewards.

2
No Pain, No Gain

In *Chapter 1*, *Evolution of a Software House*, you were introduced to ACME systems and given an insight into how they realized that there were problems with their software delivery process (severely impacting their overall product delivery capability), how they addressed these problems, evolved, and after some hard work and some time, adopted continuous delivery and DevOps ways of working. We will now be going through some of the ways in detail to identify and understand the problems you have (or not as the case may be).

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

- Achieved software delivery utopia
- You are in denial
- You don't fully understand how efficient and streamlined software delivery can actually be

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

So, let us 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 the root causes are—so that you can make the overall process more efficient and streamlined. The only way to fully understand what problems there are, is to inspect and adapt if necessary if need be adapt—as is the fundamental premise of most agile methodologies.

Before we start this inspection, I would like to go back to basics and introduce you to the elephant in the room.

Elephant in the room

Some of us have a very real and worrying ailment which blights our working lives, elephant in the room blindness—or to give it its medical name, *Pachyderm in situ vision impairedness*. We are aware of a big problem or issue which 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 may 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—can you guess?— *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.

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

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.

So, you now have your eyes open and you're ready to start removing the floral pattern from the elephant. Not quite. There's still some other things that you need to take into account before you proceed. You need to define your approach for how you go about exposing the elephant; you need to lay down some ground rules.

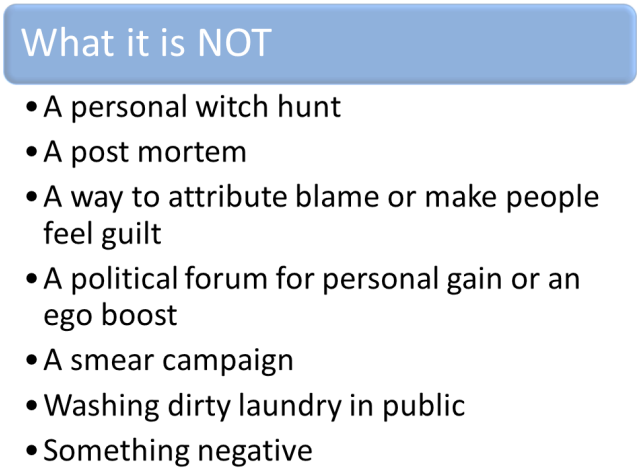
Ground rules

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 things this may include could be questions such as:

- Why things are done in certain ways?
- Who made the decision to do one thing over another?
- When exactly these decisions made?
- Who *owns* the overall product delivery process?
- Who *owns* the various steps within the process?
- What were the driving factors behind some decisions?
- Has anyone questioned the process previously and what happened?
- Why is the management not listening to us?
- Who are the stakeholders?

This dirt digging may well make some people uncomfortable and may bring to light things that produce emotive reactions—especially from those that may have originally had a hand in designing and/or implementing the very software delivery process that you are examining. Not only that, but you may need these very same people to be heavily involved in the replacement and/or refinement of it. Even if they can see and understand that the process they nurtured is broken and needs to be re-engineered, they still may have an emotional attachment to it—especially if they have been involved for a long time.

The following figure details what an investigation should not be seen as:



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.

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

Openness and honesty is the key

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 move it out of the way. You might be considering running a closed session investigation run by the management or a group of external business consultants. This may provide some level of information and data but it's a strong bet that something will be missed, some may feel intimidated and not want to disclose some pertinent piece of information or maybe not even want to get involved, someone may forget some of the details and won't have others beside them to help fill in the blanks, or you may misinterpret some of the information provided or simply take it out of context. Closed session investigations can work in certain situations and are sometimes seen—especially by the senior management—as the best way to contain a situation, quickly gather information, and stop rumors from spreading. Closed session investigations are also a hotbed for distrust, non-disclosure, disengagement, and blinkered points of view. It is therefore not recommended that you use this technique.

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 dialogue can flourish. It does not mean you have to work within a glass house and have every conversation in public and every decision made by committee (*people who live in glass houses shouldn't throw stones* and all that). What is needed is a distinct lack of secrets.

This may sound unrealistically simple and a little over the top but without openness, honesty, and transparency, people may remain guarded and you may not get all of the facts you need. You need an environment where anyone and everyone feels that they can speak their minds and more importantly, contribute. On the face of it, this may seem easy to achieve but it can be quite a challenge, especially where management types are concerned.

The sorts of things you will need to do include:

- Convincing them it is a good 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 (this may be a great number of people—I'll cover that a little later in the chapter)
- Getting them to agree to not try and drive the agenda
- Asking them to be open and honest within a wider peer group
- Ensuring that they allow subordinates to be open and honest without fear of retribution
- Asking them to keep out of the way while you work with the people who actually know what is going on and what the problems are
- Getting them to trust you

As you can imagine you may well be involved in many different kinds of delicate conversations with many people across the business.

Not only will you have challenges around convincing the management team(s) but, if you don't have the luxury of collocated teams, there is the challenge of having remote people involved in what is quite an interpersonal and interactive exercise. There are some things you could try, such as:

- Bringing members of 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 for what you'll be doing)
- If there is a time zone difference, try and pick a time when most people can be involved or run separate sessions (not ideal)

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

Let's presume that you have identified and overcome all of the challenges of getting people in the same place, you have obtained the go ahead from the management team(s), have agreed some downtime, and have a safe environment set up. You're almost ready to embark on the elephant disclosure—almost. You now need to consider how many people you actually want to involve in the inspection and to some extent how many of those you *don't* want to be involved.

Include (almost) everyone

Although you will have the best intentions and will want to include everyone involved in your software delivery process to take part in the inspection, this is neither realistic nor practical (for one thing there may not be enough space). What you do need is individuals who can actively contribute, are engaged, and who want to change things, or at least witness things change for the better. These contributors should come from all parts of the business—if they are involved in the product creation and delivery lifecycle they should be involved. This is because you need a broad set of information and data to move forward with. The keywords here are *engaged* and *contribute*.

There may be individuals who have an axe to grind or simply need a soapbox to give their personal opinion, which is fine, but this will not be constructive and may well derail the investigation process. You should not dismiss these people out of hand as they may have valuable observations to bring forward and dismissing this may foster further negativity. You should however ensure they are engaged and understand the ground rules. Of course, you may need to keep an eye on them—much like the naughty children of the class—however, you may be surprised at how much value they bring to the process.

Some people may just be too busy or simply not realize how important this exercise actually is. It's your job to ensure that the key people who fit into this category are encouraged 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.

You may have a positive problem—you just 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 could consider running multiple investigations. You shouldn't try and run any interactive session with too many people as it becomes hard to control and too many voices may increase noise rather than provide more valuable information and data.

All in all you need a good coverage of people across the business. Typically this will include some/all of the following individuals (this is a rather generic list and may or may not reflect the roles and individuals within your organization):

- Product owners/managers
- Program managers
- Software developers
- Team leads
- System administrators
- Database administrators

- Testers, QAs, and QCs
- Business analysts
- Scrum masters
- Change controllers
- Release managers
- SCM administrators

Put simply, you need to engage anyone who is 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.

Throughout this chapter you have been introduced to phrases such as *investigation*, *interactive and interpersonal session*, *elephant exposure*, and *retrospection*. These all mean pretty much the same thing: gathering information and data on how the end to end product delivery process works so that you can highlight the issues, which can then be rectified. We'll now move onto some of the ways you can gather this information and data.

Some tried and tested techniques

What follows is by no means an exhaustive list; it is simply a list of tools and techniques that could be useful. ACME systems Version 2.0 used some of these very effectively to unmask their elephant.

As previously mentioned, it is much better to have an open and interactive method for gathering information and data as it is to have a closed session. Open sessions are more engaging, collaborative, and fun. Realistically, all you'll need to run one of these sessions is the staple toolset of any agile practitioner: a big blank wall covered in paper or a large whiteboard, some sticky notes, some pens, various colored stickers, some space, and a little bit of patience.

Throughout the remainder of this chapter, we'll be referring to agile terminology so for those of you who are not fully au fait here's a simple list.

Term	Meaning
Product owner	The product owner is the primary business representative who interoperates requirements and communicates these with the team.
Scrum master	The Scrum master is responsible for maintaining the agile process/processes and the overall health of the team.

Term	Meaning
Feature	A high level business requirement (for example, a new web page to allow users to log in).
Story	A requirement (feature), which has been broken down to a level so that realistic estimates can be obtained and acceptance criteria can be defined (for example, a new login box for users to enter their details).
Backlog	A collection of stories that will be worked on at sometime in the future.
Task	A task is normally a subset of a story and consists of each small element needed to fulfill the story requirements (for example, new JavaScript function to validate for spaces in user name).
Time boxed	Is a time period of fixed length allocated to achieve some objective or goal? This method is used within agile development but can be applied to other activities (for example, retrospectives).

The following tools and techniques should help in some ways with the investigation. They are in no particular order of preference, they are simply in an order, and as stated previously, do not make up an exhaustive list. We'll start with value stream mapping.

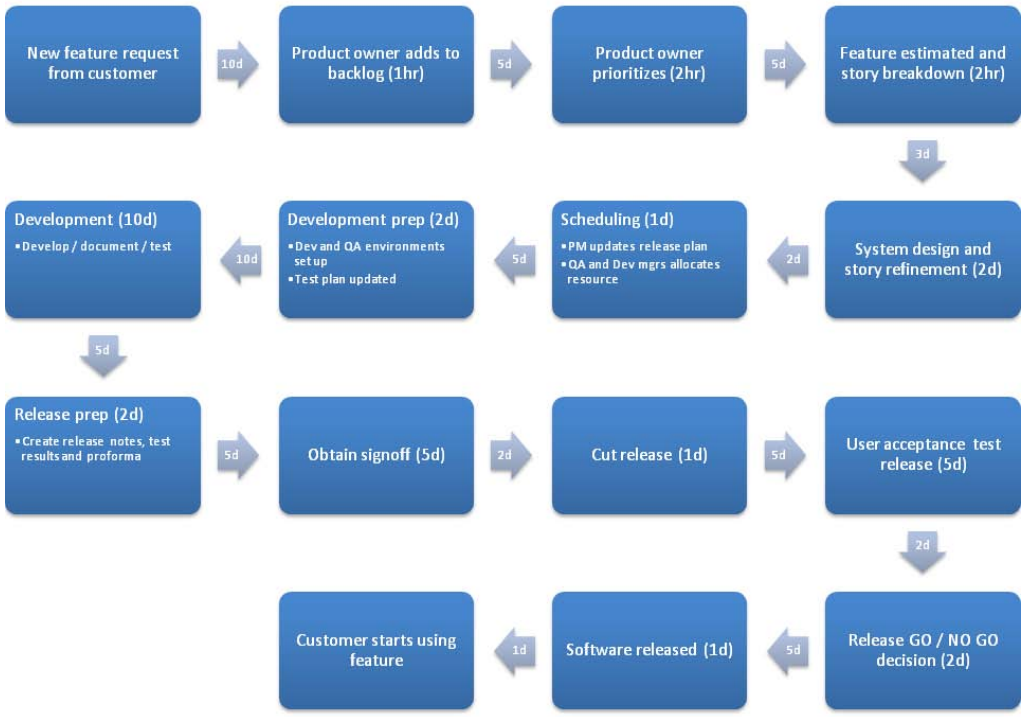
Value stream mapping

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. You may have heard of this in relation to Kanban (which is a flow-based agile software delivery methodology similar to scrum—we'll be covering that later in this book). 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 breakdown a product delivery process into a series of steps and handover points; it can also be used to help calculate efficiency rates. The overall map can be laid out and analyzed to see where bottlenecks or delays occur within the flow; in other words, which steps 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 lifecycle. Ideally a value stream map should represent a business process; however, this may be too daunting and convoluted at first. To keep things simple it may be more beneficial to pick a recent example project and/or release and break that down.

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

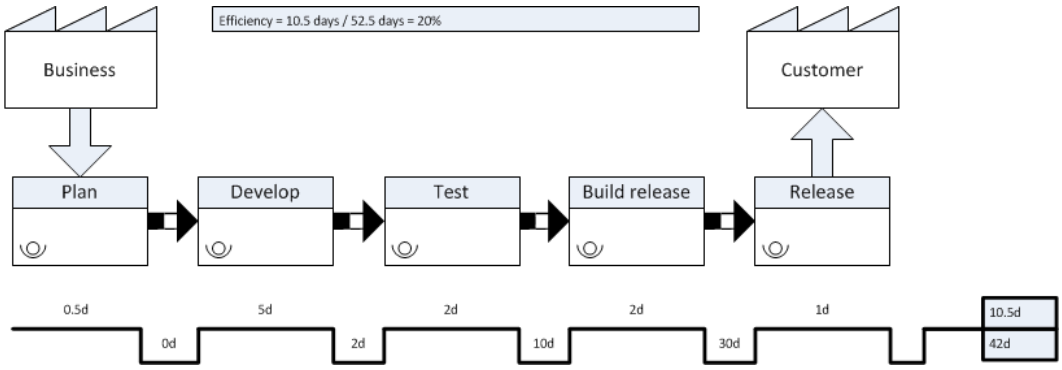


Each box represents a step within the overall process. The duration value within each box represents the working time (that is, how long it takes to go through each step). The duration value in each arrow represents the wait time between each step (that is, how long it took between each step).

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 has a cost, every delay has a cost , every 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 would 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 could 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 dead time within the flow)

Once you have a value stream map, the next challenge is then working out and planning what to do about the problems you have now highlighted. We'll be covering this in more detail in the following chapters.

The next technique we'll look at is retrospectives.

Using retrospectives

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

The remit of a retrospective is to look back over a specific period of time, or project, or release, or simply a business change and highlight what was good and bad about it. This can 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—especially in terms of time-boxing (that is, being very strict with how much time each part of the game is allocated). As with the value stream mapping technique the only tools you need are pens, paper, a whiteboard (or simply a wall), and some sticky notes.

There are many games used in retrospectives and just as many sources of information and reference materials for you to access giving you some ideas. The best thing to do is try a few and see how they work for you.

The end goal of any retrospective is to produce action points which can be taken forward as a plan for improvement. Again, we'll cover this in more detail in the next chapter.

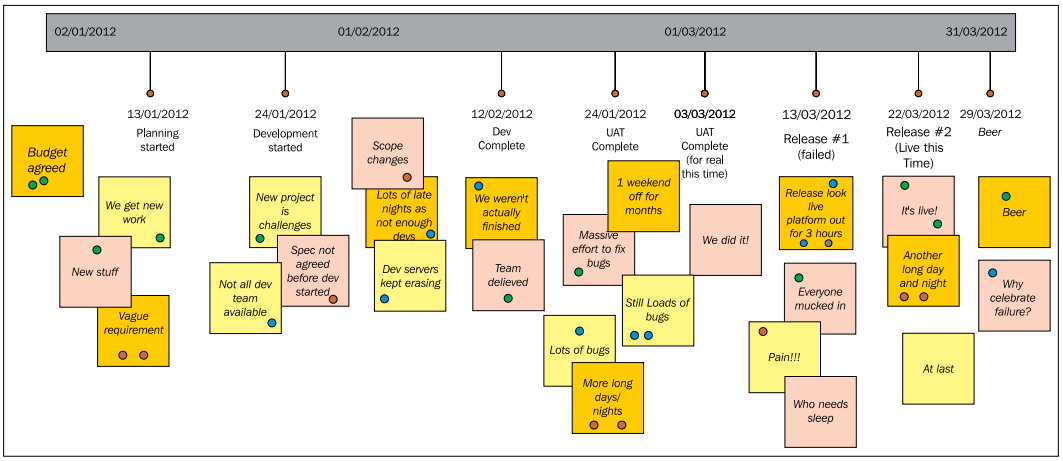
Let me introduce you to a couple of my favorite games: timeline and StoStaKee. We'll start with the timeline game.

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=glad*, *blue=sad*, and *red=mad*). From this you have an open and honest discussion about those events which provoked the most emotions and agree on actions to take forward (for example, things to stop doing, start doing, and keep doing).

Retrospectives should be time-boxed to ensure focus is retained and discussions do not drift off into the undergrowth.

The following figure depicts a typical timeline wall:

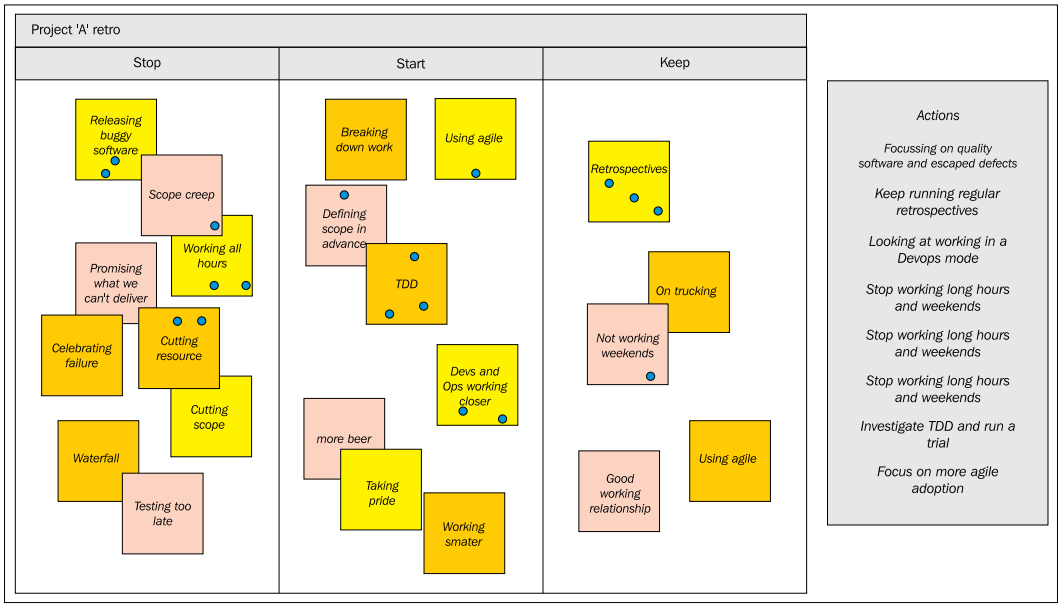


We'll now move on to the StoStaKee game.

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 discussion to ensure 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:



All in all, the techniques and tools mentioned previously are only a small subset of what is available and have proven time and time again to be most effective in investigating and more importantly understanding the issues within a product delivery process. You may feel that some do not fit in with your culture or environment, which is fine; there are plenty of other ones out there to choose from.

It may help if you engage a specialist in the field to assist you in selecting and maybe tailoring some techniques and tools for you.

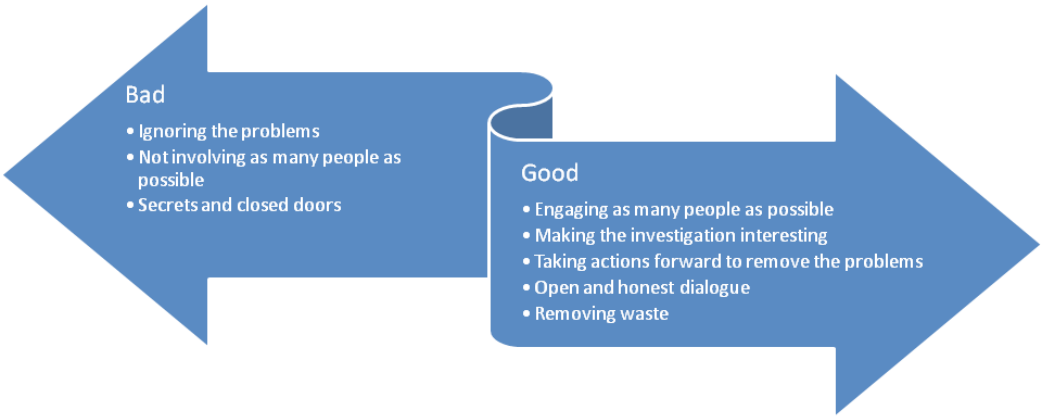
So what have we learned so far?

Summary

Throughout this chapter you have been given an insight into the following aspects:

- Understanding that your product delivery process has problems that everyone is ignoring or more likely unable to see—what we're calling the elephant in the room
- Ways of investigating the problems by engaging both management and the workforce to help identify the problems
- Some effective tools and techniques, which can help you break down the problems into easily identifiable chunks of work

The following figure sums things up quite nicely:



Now you know how to obtain valuable information and data about your problem(s) and have some much needed actions to work with. If these revolve around the waste created through long release cycles and a siloed organization you have a clear objective, which will almost certainly address the problems and deliver what the entire business needs—you need to implement continuous delivery and DevOps ways of working. All you now need to do is pull together a plan of attack to implement it, which is handy as that's what we'll be covering in the next chapter.

3
Plan of Attack

Throughout *Chapter 2*, *No Pain, No Gain*, 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:

- Waste in 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 which 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 investigation 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 goals 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 quite simple but it is not always obvious. In addition to setting the goal and vision, it is just as important what you communicate and how you do it. Do either 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 investigation 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 un-initiated 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 KISS (keep it simple stupid). You have a list of issues, which the wider business has provided for you, 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 pull together a goal and vision, which will resonate with all of the stakeholders and ensure it is something that can be delivered. It 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 soon) and formed the basis of their communication strategy. They even pulled together posters, which were stuck on the walls around the office.

Very simple tag line that anyone and everyone can understand

Yes this was an ambitious goal but they knew with a little hard work, courage, determination, and the right people involved, it was possible.

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. This may sound nice and simple but it's true to say that with a blank whiteboard and a pen you will be able to fill most of the space up with example goals. Canvas opinion from people whose judgment you trust; if they think your proposed goal is way off the mark, it might just be so. If you're lucky enough to have PR or marketing people available, canvas their opinions; this is after all something they are pretty good at. Pulling together a top level communication plan may also help to focus the message for the target audience(s).

Let's go back to the ACME CD project team to see how they went about communicating. They had a goal (deploy 10 times per day) and now had to set out the vision. This vision included a wide variety of things, some technical and some not, which could all be clearly communicated. This vision was then broken down to give an indication as to what would be addressed in what order. For those of you who are au fait with agile ways of working may spot this as *the prioritized feature backlog*, with the goal being the *epic*.

The next step was to present the goal and vision to the business and stakeholders and gain agreement that what was being proposed would address the problems and issues captured during the investigation exercise. This presentation 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.

With the vision agreed, they then went about breaking down the highest priority items of the vision (read highest priority features) into requirements (read stories), which could be easily understood and more importantly delivered.

To ensure transparency and ease of access to the goal and vision, the ACME CD 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. 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 you have a good grasp of what the business and stakeholders want, 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) who each have their own take on the terminology and vocabulary which may be used. This brings us nicely onto how you should communicate and ensure everyone involved understands what is happening.

Standardizing vocabulary and language

One small and wholly avoidable thing 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 put 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 run the tests.

There can also be a bit of a problem when you start to examine all of the different words, terminology, and TLAs (three letter acronyms) that we all use within IT. We therefore need to be mindful of the target audiences we are communicating to and with. Again the KISS method works well here. You don't necessarily have to go down to the lowest common denominator; that may be very hard to do (you could end up writing an entire book) 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 gaps and also 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	

Term	What it is	What it is not
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 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, but 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 3 to 6 month cycle to take features to the market. If all goes well, they will have to realize that a feature may be live in 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 R&D and Operations team say they will deliver something they actually deliver it, therefore, the traditional 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 size of this bubble 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 in this wider and more realistic bubble:

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

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

The benefits of a dedicated team

As with the any high profile project, it's always worth assigning dedicated resource focused on the execution and implementation of a CD and DevOps project. There is a temptation to run some projects as *skunk works* projects, which bubble along in the background and are normally staffed by like-minded individuals who have an interest but don't have the backing needed to make sweeping changes, nor the free time to dedicate to the project. Sometimes these projects simply fade away as more important projects—which do have backing and formal wide spread recognition—take the limelight and more importantly the resource. For the implementation of CD and DevOps to really work you need a dedicated team, ideally of like-minded individuals from across the business (not just developers and Operations guys), who are passionate about the subject and determined to get it working. Collocation is best, however, that's not always possible. If you have geographically diverse teams, then it may help to have some of the team members as the men/women on the ground. Their full time jobs will be to do whatever is needed to successfully implement the goal that was set out and agreed. It may not be a good idea to simply recruit a new team from outside of the business as they won't have the business domain knowledge or an established connection to the wider business. Simply put, you will need to take a number of key people out of the business for a few months to focus solely on the implementation of CD and DevOps.

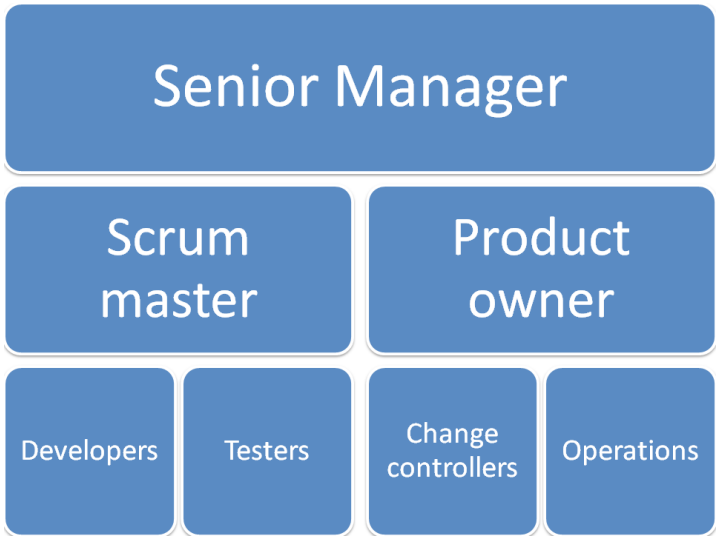
As soon as this is highlighted you will no doubt get some areas of the business take a big step back in terms of engagement—especially those areas which manage the very people you want to second onto your project. It is then down to you to cajole, beg, bargain, and barter to get the people you need. To be honest it 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 and 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. Let's take a typical discussion between you and the head of testing and QA—let's call him Chucky:

> *You: Chucky, I could really do with Steve working on the CD project with me. He knows your area of the business inside and out and from discussions we've had he seems really passionate and enthusiastic to sort things out.*
>
> *Chucky: I don't see why not, maybe one or two days each sprint would work.*
>
> *You: I'm thinking more like full time for the duration of the project.*
>
> *Chucky: I'm not too sure about that. He's a critical resource and is the only one who can deal with the developers when they deliver really buggy code, which is normally late. How many sprints are we talking?*
>
> *You: It's looking like nine to ten months.*

> *Chucky: It's out of the question. You've seen the pain we have to go through. Without Steve we'll grind to a halt.*
>
> *You: But if Steve can sort out the development process and help them implement test-driven development techniques along with all the other stuff we're looking at, the quality of the code will be the highest it's ever been. It will also arrive in small chunks that are easy to test. Added to this we have some of the Operations guys on board, so all of the environmental issues your own team highlighted should go away. All in all having Steve on board will help you and your teams more than having him stuck in a situation where he can't change or influence anything.*
>
> *Chucky: When you put it like that it's a no brainer. When do you want him to start?*

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 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.

As for how the team should be made up, it really depends on the way in which your business is set up. A typical SME would normally have something like Development, QA, Operations, and Change Management teams involved in the software release process; therefore, you should include someone from each area. Add a Scrum Master/Project Manager (that would be you) and a product owner and top it all off with a Senior Manager (someone who can act as the project sponsor and represent the team at a higher level), and you'll end up with something as shown in the following figure:



The dream team

Again, all of this depends on the way in which your business is set up; however, a dedicated team must be made up of more than just developers if they are to have credibility.

Now that you have the art of persuasion and positive influencing under your belt, you and your newly formed team need to learn the art of evangelism.

The importance of evangelism

To evangelize across an entire business all of the time 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 and your team 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 and your newly formed team need to believe in it 100 percent—if you don't then how can you expect others to?
- You and your team (or whoever is involved in the project) must practice what you preach and set the 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 and your team will have to be very, very patient. You may 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 would 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 generalistic. However, the rule of thumb is if you see their eyes glaze over, your message is not hitting home, so 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 a later chapter). If you can win them round then kudos to you and the team but don't feel dejected by one or two 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; no more, no 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 and your team 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 and 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.

Evangelism is basically 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 like a logo or some *freebies* (such as badges, mugs, mouse mats, and so on.) together to hand out. This may seem a little superfluous but as with any PR you 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.

The courage and determination required throughout the organization

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 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 above 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.

Let's break these down:

- A dedicated team assigned to the CD and DevOps project
- Changes to business process documentation and/or business process maps
- Changes to standard operating procedures
- Changes to hosting (on the assumption there is a move to virtual infrastructure)
- Tweaks to change management systems to allow for quicker and more lightweight operations
- Internal PR and marketing materials
- Enlisting the help from external specialists
- Things may slow down at the start as new ways of working bed in

These costs should not be extortionate; however, they are costs which 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 which 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 may be a good idea to do some research and/or reach out to others who have:

- Been through this transition a few times
- 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.

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*, *Some Useful Info*.

Summary

So what have we covered in this chapter? Let's recap:

- Defining a goal and vision for your CD and DevOps project is very important and needs to be simple to understand and well communicated
- Ensure that everyone understands what it is all about and is au fait with the language and terminology used
- Use online collaborative solutions, such as blogs, forums, and wikis to share information and encourage a greater degree of communication
- Making sure the business understands what it is letting itself in for in terms of the breadth of the implementation of CD and DevOps and ensuring it recognizes that it will change the business—for the better
- Why having dedicated resource is such an important thing and not something to be taken lightly
- Why effective PR, evangelism, courage, and determination are so important to the success of the project
- There will be costs, some obvious some not so, which must be taken into account before you embark on your adventure
- There are many like-minded people out there to call upon if you need it

The following diagram should sum up things for you:

Small steps

You have a team, you have a plan, you have backing, you have ways to communicate, you have recognition, you have some budget, and you have some support. You're now ready to move to the next stage—implementing the goal and vision.

4
Tools and Technical Approaches

The previous chapter focused on getting a goal, vision, and a dedicated team together to help implement CD and DevOps. Over the next couple of chapters, we will go through the steps of executing against the *plan*. Firstly we will focus on the technical side of the execution—the tools and processes you and your team should be looking to implement and/or refine.

There will be quite a lot of things to cover and take in, some of which you will need, some of which you may have in place, and some of which you may want to consider implementing later down the line. I would however recommend you read through regardless to whether there might be some small chunks of wisdom, or information at the very least, that you can adapt or adopt to better fit your requirements. Quite a bit of this chapter is focused on software engineering (that is, the Dev side of the DevOps partnership) but bear with me as some of the points covered are as relevant to system operations as they are to software engineering.

It is worth pointing out that the tools and processes mentioned are not mutually exclusive—it is not a case of all or nothing; you just need to pick what works for you. That said there is a logical order and dependency to some of the things covered over the next chapter or two but it's down to you to decide what is viable or not.

We will start with some good engineering practices.

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 may be extremely minimal. Why, I hear you ask, do I need to know how a developer does his/her 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 to 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. So 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* individuals.

CD and DevOps is based upon a premise that quality software can be developed, built, tested, and shipped very quickly many times in quick succession—ideally we're talking hours or days at the most. It's normally the case that the first three are a given for most agile software development projects; it's the shipping bit that takes the time—as you will have no doubt found out during your investigations. For more waterfall style software development projects there is most probably some waste in the *develop* > *build* > *test* cycle of the process. Let's go back to some fundamentals in terms of software engineering, which may help:

- Always use source control
- Commit small code changes frequently
- Do not make code overly complex and keep it documented
- If you have automated tests, run them very frequently
- If you have a continuous integration (CI) solution controlling your build and test suite, run it very frequently
- Use regular code reviews
- Do not be afraid of having tests that fail or others finding fault in your code

These are quite simple rules and as stated previously most software engineers who work on modern agile software development projects will see the above as common sense and common practice. I say most because there may still be some old school code cutters who believe that they are exempt because they have been doing the same thing for 20 years. What is more worrying is the fact that there are many software engineers who do not have the luxury of following the above simple rules. Either they do not have the opportunity, they are not aware of them, or simply work within an environment which does not allow for or believe in the above (yes these places do exist).

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

There are many different flavors, versions, and solutions available for source control (sometimes referred to as SCM or version control systems), both commercial and open source. Therefore, there are no excuses not to use source control. If your code is in source control it is versioned (that is, there is a history of every change that has been made from year dot), it is available to anyone who has access to the source control system, it is secure, and it is (normally) backed up so you don't lose any of it.

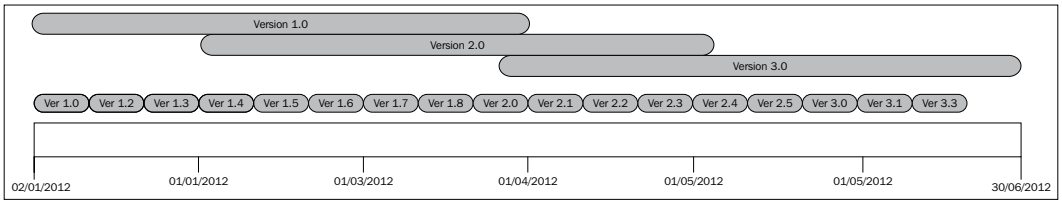
There are books and reference materials aplenty available, which cover this subject in much more depth, so I will not dwell on it here. Suffice to say, if you do not have a source control solution then implement one. Now!

Source control should not be restricted to software source code. You can (and should) utilize source control for anything within your system that can be changed. This includes things such as system configuration, start-up scripts, server configuration, network configuration, and so on. In short, if any part of the overall platform can be represented and stored as a text file, it should be stored within source control.

A source control solution is a very valuable 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. It sounds overly simplistic but it is also very true. The following diagram gives some indications into how this could look:



Large releases versus small incremental releases

You might not currently have the luxury of shipping your code very frequently but that is no excuse for not using best practice. Once you have CD and DevOps up and running you'll have to work in this mode so why not start getting used to it.

Developing in small and frequent changes can also help with reducing complexity and maintaining quality. Small changes also make merging of code easier to manage as you only have a small amount of changed code to bring together; this can also help you if you are moving towards the engineering utopia of developing from trunk all of the time.

This practice should 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 (such as memory allocation to a virtual server) 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.

Working with small, incremental changes is a very beneficial practice to follow, as is ensuring you have a good understanding of how breaking changes can affect your platform.

Never break your consumer

Your software 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 some predefined interface spec (sometimes called a service contract).

A simple example would be a web page that utilizes a PHP library to return content to render and display a user's address details. In this scenario, the web page is the *consumer* and the PHP library is the *provider*. If the PHP library originally returned four pieces of data but was changed to provide five, the consumer would not know how to handle this and may well throw an error or worse still, simply crash.

As the complexity of software platform increases, it is sometimes very difficult to spot which *provider* has changed and is causing one of the many *consumers* to fail—especially when you consider that a *consumer* may also be a *provider* to another *consumer* higher up the stack (that is, a PHP shared library, which *consumes* from a database then *provides* the said data to a web page, which then *consumes* it and *provides* that data to a JavaScript client, and so on). Some sort of impact analysis may well help map this out but unless you can map out your entire platform into one easy-to-understand format that is consistently up to date, it's going to be very difficult.

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 which will cause the *consumer* to fail.

Sometimes breaking changes cannot be avoided (for example, the interface spec between components has to change to accommodate new functionality). However, this should be the exception rather than the rule and you should have a strategy planned out to cater for this. An example strategy would be to accommodate side by side versioning, which will allow you to run more than one version of a software asset at the same time.

There may be times when the consumer/provider relationship fails as the person or team working on the provider is unaware of the relationship. This can be very true of providers within the system operations area. To overcome this, or at least 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 methodologies in use today but all of them revolve around some form of collaborative ways of working, be that pair programming or simply using code reviews within the development lifecycle itself. I cannot express enough the importance of sharing your code with others. Even the best software engineer (or systems admin) on the planet is human and they can/will make mistakes. If you think your code is precious and don't want to share it with anyone else you will create bugs and you will take longer to overcome small mistakes, which can cause you hours of head scratching or worse still have an adverse impact on your customers.

If you are confident that your code is of the highest quality and can stand up to scrutiny, then do not hide it away. If you are not confident, then sharing with your peer group will help. One thing to point out here is that your peer group should not be exclusively made up of software engineers; the Operations team should also be included in this process. It may seem strange as they may not be able to actually read your code (although you may be surprised how many system admins can read code), but they know how the live platform operates and may be able to provide some valuable input and/or ask some pertinent questions.

All in all the majority of the world's highest quality software is built in a collaborative way so there are no excuses that you should not be doing the same.

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 not to share system configuration changes.

One normally unforeseen advantage of peer working is that if your code (or configuration 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 fail.

Fail fast and often

Failing fast and often may seem counterintuitive but it's a very good ethos to work to. If a bug is created but it is not spotted until it has gone live, the cost of rectifying the bug is high (it could be a completely new release), not to mention the impact it could have on your reputation and possibly your revenue. Finding bugs early on is a must.

Test driven development (TDD) is based upon the principle of finding fault with software very early on in the process. Before code development begins, tests are written to cover the majority of the use cases the software has to cater to. As the code is being written, these tests can be run again and again. If the code fails at this point that is a good thing as the only person impacted is the software engineer rather than your customers.

This may sound strange—especially for the managers out there—but if bugs are found early on you should not make a big thing of it and you should not chastise people. There may 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.

One of the things that can sometimes scupper implementing engineering practices such as TDD is the size and complexity of the software platform itself. It may be a very daunting task to retrospectively implement a test suite for a software system that was not built around these principles. If this is the case then it may be wise to start small and build up.

Automated build and testing

Following the fail fast and TDD approach provides quick feedback to the engineer as to whether the change they have made works (or not as the case may be). What is also very helpful is to understand if the code they have written actually builds/compiles consistently. You could of course use manual processes to achieve this but that can be cumbersome, inconsistent, prone to error, slow, and not always fully repeatable.

Implementing automation will help speed things up, keep things consistent, and above all provide confidence. If you are running the same steps over and over again and getting the same results, it's a strong bet that it works and you can trust it. It is therefore plausible that if you change one thing within your platform and the previously working process fails, there is a very good chance that the change has broken something.

There are plenty of tools available for building/compiling code. All of which do pretty much the same thing—ensure the code is written correctly, ensure all of the external references are available, and if so create a binary to run.

As for the tools and technologies you can use to create automated tests, again there are many different flavors and solutions available. It is worth taking your time to investigate some of what is on offer (or ask the engineering teams what they use/would like to use). With a little R&D you'll be able to get something up and running quite quickly.

Staying with automated testing there is one thing that puts people off; that being the fact that it can be quite daunting. How much of the platform do you cover? How do you replicate users in the real world? Where do you start? There is no straightforward answer other than keep it simple (KISS). Start with mapping out the primary use cases and create automated tests to cover them. You can always refine these or add more as you go along.

One small snippet of advice regarding automated tests is not to over complicate your test data. 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 have the test script create and more importantly tear down the data it needs during execution of the test itself. If you do not follow this, you will end up with stale data that may well become out of date quite quickly.

So we have some automation to build and test your software components but how do we set it up so that we can run it all as and when we need it? That is where continuous integration solutions come into play.

Continuous integration

Continuous integration—or CI as it's more commonly known as—is a tried and tested method of ensuring the software asset, which is being developed 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 on each commit to the source control system if not more frequently).

There are very many mature CI solutions available—some free, some commercial—so just like a source control system there are no excuses not to implement and use CI.

CI solutions are basically software solutions that allow you to run your automated scripts from within *CI jobs* when certain events occur (commits to source control, every ten minutes, overnight, and so on). These 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 and run them.

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 you can add the executable to your binary repository or if something fails, e-mail the results to the engineering team. You can even build dashboards or radiators so 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.

The advantages of CI for systems operations changes are not as obvious but they can help a great deal in terms of trying out changes without impacting the production platform. For example, let us presume that you have a CI solution that is running many overnight automated tests against an isolated test environment. The tests have been *green* for a few days so you are confident that everything is as it should be. You then make a server configuration change and re-run the CI suite, which then fails. The only change has been the server configuration, therefore it must have had an adverse impact.

Implementing CI is no small challenge—especially if you have nothing in terms of automation to start with. However, CI is a very powerful tool and vastly reduces that overhead and risk of using manual methods for building and testing system changes.

We will now look at some other technical challenges that you may need to overcome. Let's start with system architecture.

Architectural approaches

A vast number of software platforms have evolved over many years and some can be rather complex and cumbersome to maintain or advance. There are as many different ways to build and design software platforms as there are development languages. If you have to maintain or advance what many would class as a *legacy* product (which may be one huge blob of executable code that must be released as one single bundle), then you may be severely restricted when it comes to CD and DevOps. That said, it might not be the fault of the system and/or architectural design, rather the processes that are in place to release the software. Even the most integrated and closely coupled software platform is actually made up of many small components all talking to each other.

If you take a step back and look at your platform you'll most probably find you could actually spilt it down (or at least most of it) into small manageable chunks (shared libraries, different layers of technology, and so on), which can be built and deployed independently and more importantly can be released frequently.

Component based architecture

If you are lucky enough to have the opportunity to reengineer your platform—as did ACME systems—then you should take time to consider the best approach for your needs. Ideally, you should look at a technology or an architectural approach that allows the platform to be broken down into small discrete modules or components that are loosely coupled.

Something similar to a web services or service orientated architecture (SOA) is a good start. Going down this route you have the advantages of small, discreet software components that can be developed and more importantly released separately, which goes some way to realizing the aforementioned principles of CD and DevOps; CD and DevOps is based upon a premise that quality software can be developed, built, tested, and shipped very quickly many times in quick succession.

Having a component-based architecture allows for small and frequent changes to be released, it can also simplify the physical implementation and platform infrastructure. If you currently have one or two huge bits of code, you have to have one or two hulking great big servers to run them. Breaking this down can allow for greater freedom in the way the infrastructure needed to run the overall platform is designed and implemented.

There is a mountain of options and information available to determine the best approach for your current and future needs. Suffice to say if you can move towards a component-based architecture, the pain and overhead of releasing will be a thing of the past.

One important thing to note here is that having a component-based architecture but still using the *clump it all into one big release* approach, will actually be more painful than releasing a legacy type platform. It is therefore very important that you take this into account.

Let's have a look at another possible solution that may help if you're not lucky enough to be able to reengineer your platform, or already have a component-based architecture but there are some hard dependencies which cause deployment pain.

Layers of abstraction

If you have quite complex dependencies throughout your platform, it may help to try and separate your software assets by using some form of abstraction. This technique should assist in removing or at least reducing hard dependencies within your platform. If for example, you have two components that have to be deployed together as they have been hardwired in such a way that deploying one without the other would cause a platform outage, then you're going to struggle to follow the *small incremental changes* method—not to mention the fact that you will be hard pressed to release without downtime.

There are plenty of design patterns available which can give some good ways of achieving this, but at the very least it is a good practice to remove close dependencies wherever possible so that you don't end up with clumps of assets which need to be deployed together.

A simple example would be the use of databases. If you have some code running on your platform, which accesses a database table directly and a new column is added to the database table, you may also have to recompile and re-release the software asset along with the database change, even if there are no code changes. To get around this you could look at using SQL VIEWS, which adds an abstraction layer between the code and the database. That way if there are changes to the database but the VIEW doesn't change, there's no need to recompile and re-release the code.

You should at least spend some time looking at areas of the platform, which are closely coupled and see if some sort of abstraction layer can be placed between various parts to reduce the need for releasing big clumps of the platform together.

Let's assume you have seen the light and have small loosely coupled components, written using best practice and you're ready to move to the next stage in your software engineering evolution. What you now need to consider is how many different environments you need to deploy to so that you can ensure a given change will not impact the production platform and your users.

How many environments is enough?

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

The ideal number of environments is two—one for Development and one for Production. This may sound like an accident waiting to happen but many small businesses manage fine with such a set up. As a business grows so does the need to be risk averse, hence the potential for multiple environments.

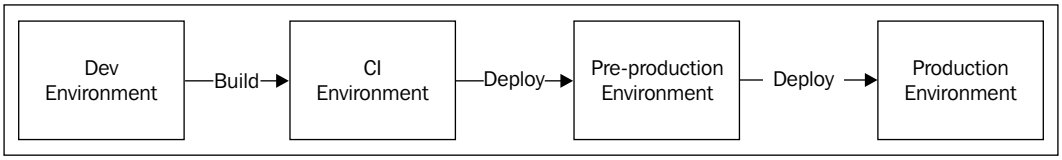
When ACME systems started out, two environments were sufficient. As they grew, so did the need for more environments and they ended up with multiple development environments (one for each engineering team), an integration environment, a performance testing environment, a pre-live deployment staging environment, and of course production environments. They also ended up with an entire team of people whose job was to keep these all running—actually they ended up with two, one to look after the engineering and testing environments and one to look after the production environments. Not ideal.

Of course with vitalization technologies now matured and used by anyone and everyone, setting up and running hundreds of servers is not as much as an overhead as you would think, however it's the challenge of keeping everything in line that is massive—versions of software, O/S patch levels, network configuration, and so on. There is also the not uncommon issues around having production environment(s) in a less accessible area (maybe a secure DC) or even managed by a third party.

When ACME systems reviewed the environments required for CD and DevOps they settled on the following as being sufficient for their needs:

- Development environments—cut down versions of the platform with only a few other platform components
- CI environment—the place where the software is built and all automated tests are run on a regular basis
- Pre-production environment—for the occasional spot check/UAT (occasional being the operative word)
- Production environment—this is where all the action takes place

The following diagram depicts the environments used:



ACME systems 3.0 environment setup

What was sufficient for ACME systems may not fit your needs exactly, but as you can see it's quite a simple setup.

Once you have the environments you need, the next challenge is ensuring you are deploying the same thing to all of them.

Using the same binary across all environments

When a software asset is *complete* it has normally been built/compiled into an executable. What you need to ensure is that this binary is only built once for a given release/deployment and that the self-same unchanged binary is used in all environments—including production. This may sound like common sense but sometimes this is overlooked or simply cannot be done.

For example, let's take the credentials for a database server. Some would say that because this is top secret it should not be available to all. Some would also say that this should be *baked* into the binary itself at compile time. This is all well and good but that means you would have to have the same credentials set up in all environments, including the completely open development environment. To get around this you could create more than one binary, one for each environment however, this would mean that you would be testing different versions of the software.

Ideally you should have this kind of configuration held in a startup script or system configuration file (which of course is under version control) and have the software load it up at runtime. That way you have one binary in all environments and a unique configuration per environment. If you bundle up your component into an installation package (say an `EXE` or an `RPM`), you could include the configuration within that to ensure it is all packaged up together—you will still be installing the same binary.

As with source code, binaries should be versioned and stored in a repository. Again, there are many such solutions available so there is no excuse not to use one.

We have looked at environments and using the same binary across all of them. Let us now look at how engineers can develop and test against components that are currently running in the production environment.

Develop against a like live environment

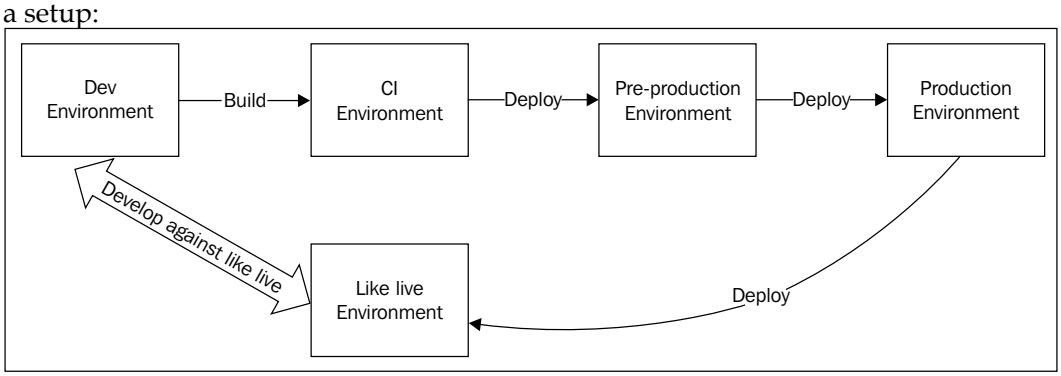
There are many ways to ensure one version of software works, or integrates, with other parts of your platform but by far the easiest is to actually develop against an environment, which contains live versions of code.

This means that you are ensuring that the dependencies you expect to be available to your software within the production environment are actually there and function as you expect. The utopia would of course be to develop against the production environment but this is very risky and the possibility that you could cause outage—albeit inadvertently—is quite high. Therefore having another environment set up, which contains all of the versions of code that is running in the production environment, is the next best thing.

You may be thinking that developing against a like live environment is somewhat overkill and you may be wondering why you do not simply develop against the versions of software that reside in the CI environment. There is a simple answer; you have no firm idea which of the versions in the CI environment will be live before you. This is especially true if someone is testing out a breaking change. Your best and safest bet is to develop against what is currently live.

This *like live* environment only needs to be like live in terms of software (and infrastructure) versions. If you can populate it with live data then that would be good but the reality is that you would need something as big as production in terms of storage and so on, which is costly.

The following diagram gives an overview of how ACME systems implemented such a setup:



The like live environment used by ACME systems 3.0

As you can see, the like live environment is tagged onto the end of the deployment pipeline. It is on the end for a reason; you only want to deploy to this environment once the deployment to live is successful.

Alternatively you could look at virtualization on the desktop, whereby 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).

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, those being how you actually take the fully built and tested software component through the environments. Here is where CD tooling comes into play.

CD tooling

There is another collection of tools which may not be as readily available to you as the aforementioned tools (automated build and testing, CI, and so on). These are the tools that you will use to control and orchestrate the act of deploying your software components to the various environments you have. That is not to say that there are no tools available, it's simply the case that there are not that many out there and those that are, may not completely fit your needs, requirements, or ways of working. You could of course take on a tool (or a collection of tools) and change your ways of working to fit but this may have a huge overhead. It may be far simpler—and more beneficial in terms of future proofing your CD process—to build your own.

The tooling you choose (or build) will be used day in and day out and will be heavily relied upon, so you had better make sure it is very good. It should also be very easy to use and simple to operate—ideally *one click to deploy* simple.

When ACME systems looked into this, they decided that what was available did not fit the bill so they decided to build their own tooling that fitted their needs at the time and could be expanded and extended as the CD and DevOps adoption matured. They of course used all of the engineering best practices mentioned previously to build it and even went so far as to treat the solution they created as an open source type product, meaning anyone within the ACME engineering team—not just the dedicated team members—could advance and enhance it.

The things that ACME systems considered—and you may well need to consider when looking at tooling of this type—were as follows:

- Can it deploy the same binary to multiple environments?
- Can it access the binary and source repositories?
- Can it remotely invoke and control the installation process on the server it's been deployed to?
- Is it capable of deploying database changes?
- Can it manage or at least reference environment/server specific configuration and deploy that along with the binary?
- Does it have 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, 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 do manage to find a tool which fits all of these, then congratulations. If not then you had better get busy coding or at least bolting together a number of tools to cover your needs.

One of the considerations noted above is automated provisioning. Let's look into what this means.

Automated provisioning

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

Provisioning is nothing new; as long as the likes of Amazon have been providing their *cloud servers* you have been able to provision what you want when you need it (for a price). Having 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 ever, there are many industry buzzwords floating around to complicate this sort of activity—infrastructure-as-a-service (IaaS), platform-as-a-service (PaaS)—but what it boils down to is being able to programmatically *talk* to a system, tell it what you want in terms of spec, operating system, configuration, and so on, and get it to spit one out the other end.

In essence you have the fully tested software asset, you have the recipe for your environment/server configuration (this is held within the CD tooling), so the act of deployment is quite simple:

- Provision a server your software asset needs to run on
- Deploy the software asset onto the server
- Install it
- Add the server to your platform
- Start using it

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 do 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 comes with large releases of software (legacy or otherwise) is the unforgivable need to take some or the entire platform offline while the release happens. Yes I did say unforgivable because that's 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 may even shop around to find a competitor who does not have downtime.

Okay so this is a bit on the dark and negative side but that 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 do have to implement breaking changes implement them gradually rather than with a big bang (for example, start at the bottom of the stack and work up)

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

One thing to point out which may not be obvious is that it's not just the production environment which should have maximum uptime. Any environment which you are reliant 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.

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 pretty good for monitoring infrastructure and servers, and Graphite is pretty good at collecting application metrics. Unless you can tie the two together into a 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 come to that) is coping and functioning. You will be surprised how much very valuable data you can get and how it can direct your engineering 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 to work correctly. As change is being made to production (software or infrastructure) both the Dev and Ops side 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 production platform. If you're using some *graph over time* solution you can get your CD tools to add a *spike* or a marker to the graph when a deployment takes place. You can then visually see the impact (or not) of the change.

Up until this point we have mainly focused on technical solutions for the implementation of CD and DevOps. These solutions may 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 may well end up back where you started. You may 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.

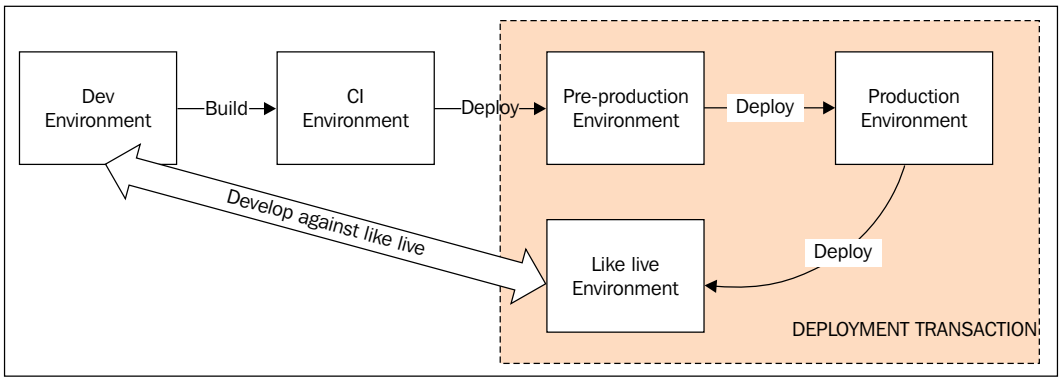
You can try to map this out and build into the tooling rules to restrict or at least minimize these moving targets but the rules may end up more complex than the original dependencies themselves. Or you could simply agree on a process whereby only one change happens at any given point in time. To feed 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 build badger*. If you had the build badger you were changing production.
- They implemented a simple prioritized queue system using a whiteboard and a pen. Each morning whoever wanted to make a deployment would come along to the deployment stand-up and agree with everyone there 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 of the teams involved.

The following diagram should give you some idea of what the deployment transaction covered—in terms of a deployment:



The deployment transaction

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

- Use collaborative tools for real-time communication between everyone (that is, 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, look into simple ways of rolling back, such as simply 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.

Summary

As stated at the beginning of this chapter there is a lot to cover and a lot to take in. Some of it is relevant to you now and some of it will be relevant for the future. Here's what we have covered:

- Using engineering best practices is a must and it's not too hard to do
- There are no excuses to not implement source control, automation, or CI
- If you have a reliable and repeatable way to create small, fully tested software assets frequently, the act of deployment becomes much less painful
- You must ensure that you use the same binary across all environments and you don't need 10s or 100s of environments to effectively test your software and platform
- Not all of the tools you need are technical and sometimes a small manual process or activity is more useful and powerful than a fully automated tool
- If your platform supports virtualization, then use it
- If you can't find CD tooling that matches your needs, build some
- There are no excuses for taking your production platform (or any environment your CD process is dependent on) offline to upgrade it.
- Monitor, monitor, monitor

As you can see there is quite a bit for you and your team to do, to implement and to teach others. It's not all technical either; there is a large chunk of simply convincing people to adopt and use the tools and processes laid out above. And that's the easy part. In the next chapter, we'll be covering the human side of DevOps, the culture and behaviors that you'll need to work on.

5
Culture and Behaviors

The previous chapter focused on the tools, some technical and some less so, that you will need in your CD and DevOps toolbox. You may not need them all initially but having them there when you do is a good idea. What we will now do is move our attention to what some call the *soft and fluffy* side of implementing CD and DevOps. We will cover things such as:

- 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
- Building trust
- Rewarding success in the right way
- Instilling a sense that change is good and not risky
- How good PR can help

As with the previous chapter there is quite a bit to cover, some of which will be relevant to you now, and some relevant in the future, but all of it is relevant.

As with the previous chapter, it is worth pointing out that the topics covered are not mutually exclusive—it is not a case of all or nothing—you just need to pick what works for you but be aware of them all.

We'll start with something which on the face of it should be quite simple: encouraging open, honest, and courageous dialogue.

Open, honest, and courageous dialogue

Apart from sounding like something taken directly out of a management training manual, what does open, honest, and courageous dialogue 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 fear of ridicule or retribution. Some working environments do not encourage this way of working or worse still it is actively discouraged. If this is the situation then you have some challenges to overcome, especially with the management and HR types. They are normally the ones that define and enforce the guidelines and policies under which every employee should operate.

Openness and honesty

Openness and honesty are key factors to ensure that the implementation of CD and DevOps works. Without this type of environment it's going to be very difficult to break down the barriers and implement the much needed changes throughout your organization. You have already engaged the majority of the business during the investigation stages to obtain honest feedback about the current situation and you now need to ensure 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, which they can use to share their thoughts. Not only that, they need to be actively encouraged to do so, which can be quite a challenge.

We did previously mention the use of online collaboration tools, which can be very effective for this type of problem, especially something like a real-time forum solution or a group chat system. There of course has to be some form of etiquette or guidelines as to what is acceptable and what is not, and common sense should prevail. What should not prevail is a heavy handed policing or moderation of the content. This will discourage openness and honesty and ultimately make the solution redundant.

As you go through the implementation of CD and DevOps it is extremely important that you and your team also have regular and truthful feedback from all concerned (especially those working at the coal face), in terms of what is working and more importantly what is not. As with any agile methodology there has to be a quick feedback loop for you to inspect and adapt to. Again the far simplest and most effective way is to walk around and ask people. You could of course use some sort of lightweight survey solution if walking and talking is not wholly viable.

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 are going to 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 software delivery process. Or you may have individuals whose views are at odds with 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 black mark on their record. They need to be given a DMZ (de-militarized zone) 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 needs be, to ensure that there is a forum for this type of dialogue as it is very important. The content may not be enlightening but if you have a number of people all saying the same thing, then there is something that needs to be addressed. At the very least you could work with the management types to implement some sort of amnesty or maybe a way for anonymous feedback to be collected.

If you work to the rule that *no news is good news* then you are misguided and you will find failure knocking on the door when you least expect it.

Let's be very open and honest about how easy it is going to be to implement and embed 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 work), it works very effectively. You will find that once openness and honesty are embedded into the normal ways of working, things really start coming together. The need for courageous dialogue will also diminish as everyone and anyone should know what is happening across the business (within reason).

Let's summarize what we have covered so far:

Good	Bad
Allow freedom of expression	Having a closed and secretive environment
Encourage anyone and everyone to have their say	Ignoring what you are being told
Be patient with the quiet ones as it will take a bit longer for them to open up	Using feedback in a negative or nefarious way
Ensure management and HR understand why this is needed	Being impatient
Refine existing surveys or create new ones	Do as I say not as I do
Get management to contribute	

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

The physical environment

Some of you may 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 to people you collaborate with. The reality is that many of you may not be not so lucky and may have teams physically separated by office walls, flights of stairs, the dreaded cubicles, 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 could look at to remove some of these:

- Keep the office doors open or, if possible, remove them altogether.
- Set aside an area for communal gatherings (normally in the vicinity of coffee) 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 table football competition within the office.
- If you use scrum (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.
- See if some of the partition walls can be removed.
- If you have cubicles, remove 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 if an office move around is possible to get people working closer together
- Implement some sort of collaborative forum/messaging/chat solution (for example, IRC, campfire, or similar), which everyone can use and have access to.
- 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 yourself. The end game here is to start removing the barriers, both virtual and physical.

We'll now move on from the seemingly simple subject of openness and honesty and move into the seemingly as 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 may 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—maybe you're busy building tools or focusing on certain areas of the existing process, which 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 onto them; they need to feel it's their idea. Some simple examples could be:

- If you have implemented an online forum or group chat solution, encourage everyone to use this rather than e-mail—maybe 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 five 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 fashion talking to each other

- Even if you don't follow a scrum methodology use the daily stand-up technique across the board—maybe even mix it up so people can move between the stand-ups
- Ensure you and your team mingle and keep in open discussions with all teams—you never know, you may hear something, which another person has also been discussing and you can act as the DevOps matchmakers

As well as 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.

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 you'll start to see them: more general conversation 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, more background noise as people talk more or even share the joke of the day.

Some subtle (or sometimes not so subtle) PR may help: posters around the office, coffee mugs, prizes for most collaborative project team, and so on. Anything to keep collaboration in sight and mind.

Let's leave collaboration for now and together move onto 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 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—that and the fine wine, fast cars, and the international jet-setter lifestyle (maybe that's stretching things a bit too far).

That isn't to say that they can all go off and do what they want; there is still a product to deliver. 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 may be different ways of working or product delivery methodologies that come to light, which you could and should be considering.

TDD, scrum, KanBan, and so on, 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 an idea or concept.

Innovation can also increase risk, new things always do; therefore the engineering teams must understand that with freedom comes responsibility, ownership and accountability for the *new stuff* they produce and/or implement.

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 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.

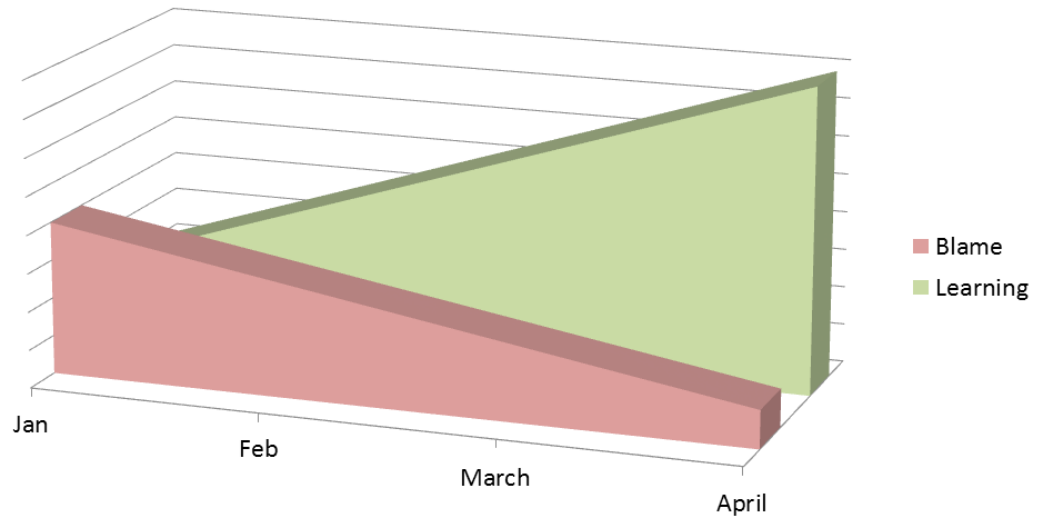
The blame culture

As previously covered, 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 that 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 have covered so far.

Blame slow, learn quickly

In a commercial business this may sound strange and may be seen as giving out the wrong message (for example, it may 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 may 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 may not be easy for some—especially the managers who have built up the reputation of being Mr. or Mrs. Shouty. Sometimes they will adapt and sometimes they may simply step out of the way of progress—as the groundswell gains momentum they will have little choice to do one or the other.

Let's again summarize this:

Good	Bad
Accepting accidents will happen	Pointing fingers
Encouraging a fail fast, learn quickly culture	Calling out an individual's failings
Encouraging accountability	Blaming before all of the facts are known
Encouraging the open and honest sharing of lessons learned	Halting progress
Not making a big thing of issues	
Focusing on individuals who don't exhibit good behaviors	

All in all, 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. If the concern of management is that everyone will get away with murder (figuratively speaking), then ask them to consider what is best—blaming those who want to do their best or focusing on those who don't? If everyone is blamed no matter what, you'll be hard pressed to pick one out from another.

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 a 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 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 like that. Trust is earned over time through people's actions.

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

Building trust between a traditional Development and Operations team 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 investigate issues when they occur, and the Operations team don't trust that the developers won't bring the 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 sort 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 for forging a good trust-based relationship between two or more factions; however, the following techniques proved to work for ACME systems and may well help you:

- If you are arranging offsite CD training, ensure that you get a mix of software and Operations engineers to attend and ensure they are in the same hotel. You would 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 DevOps Days), again make sure there's a mix of Devs and Ops 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 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 Ops for a month and vice versa). This could also include management roles.

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

Rewarding good behaviors and success

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 get the release out of the door? On the face of it, this is good business practice and management 101, 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 failure—because not delivering what the business asked for and/or impacting your customers is a failure—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 one of those types of businesses, 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 the failure. 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 may 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 have decided to put some extra effort in to overcome some long outstanding technical debt or implement some labor saving tools to speed things up then that is completely different and you should be looking at specific rewards for those 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 *releases* anymore—they are happening too quickly to notice each one—therefore you need to look at other ways to reward. For example, you could look at throwing a party when a business milestone is hit (such as, when you reach the next millionth customer) or when a new product launches.

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 other good behaviors previously mentioned (openness and honesty, innovation, accountability, and so on). This can be quite a shift for some businesses and may even need a new rewards system to be put into place.

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

Recognizing how different teams are incentivized can have an impact

There is a simple and obvious fact that some people may not instantly realize but it is something that is very real and very widespread throughout the entire IT industry. This fact is that Product Development teams are incentivized to deliver change whereas Operations teams are incentivized to ensure stability and system uptime, thus discouraging change.

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 lots 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 could 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 may 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 may be ignored as focus will remain upon 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 would be ideal as it removes all of the conflict between the engineering teams and refocuses them on what is important, that being, delivering products customers want and need.	The reality is that it would be quite difficult to get full agreement and get it in place quickly, especially in a corporate environment. That doesn't mean it's not something worth pursuing.

Whatever you do in regards to incentivizing 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 15-year old code, an upgrade to an operating system, 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 getting the other parts of the business that are, by their very nature, risk averse. I'm thinking here of the QA teams, 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 and 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 in the change to post regular blog entries detailing what they are doing and 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 and present it to those previously engaged

Okay so the above may be overkill and if all of it were implemented for each and every change it would take up a vast amount of everyone's time but it does serve a purpose; it proves to the business that change is good and risks can be controlled and managed. I would suggest you look at running this at least once or twice to build confidence. It will also 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 be more diligent.

The above may seem like overkill but this is nothing compared to how some organizations currently function; changes are fully documented and risk assessed, progress meetings are held, project progress is publicized, 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. 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 like that.

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 the CD and DevOps adoption. Being highly visible is key to breaking down barriers and ensuring everyone and anyone is aware of what is going on.

Being highly visible about what you are doing and how you are doing it

As we previously covered, being secretive about what you are doing and how you are doing it is not conducive to building an open, honest, and trust-based working environment or culture. If everyone and anyone 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 big screens around the office to display stats, facts, and figures. You may well have something like this set up already but I suspect these monitors display very technical information—system stats, CPU graphs, alerts, and so on. I would also suspect that most of these reside in the technical team areas (Development, Ops, and so on). This is not a bad thing it's just very specialized, and those of a non-technical nature may well ignore them.

What you really need is to complement this information with very simple, easy to read and understand data related to your CD process. You should be looking at displaying the following kind 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 the progress of the current release going through the process and who initiated it
- Production system availability (current and historical)
- Latest business information such as share price, active user numbers, the number of outstanding customer care tickets, and so on

All in all you need to display and advertise information and data that is business relevant rather than simply focusing on technical facts and figures.

Having this information visible as you progress through your adoption and implementation of CD and DevOps will also provide proof that things are improving.

We're now at the end of this chapter, so let's review what we have covered and what we have learned.

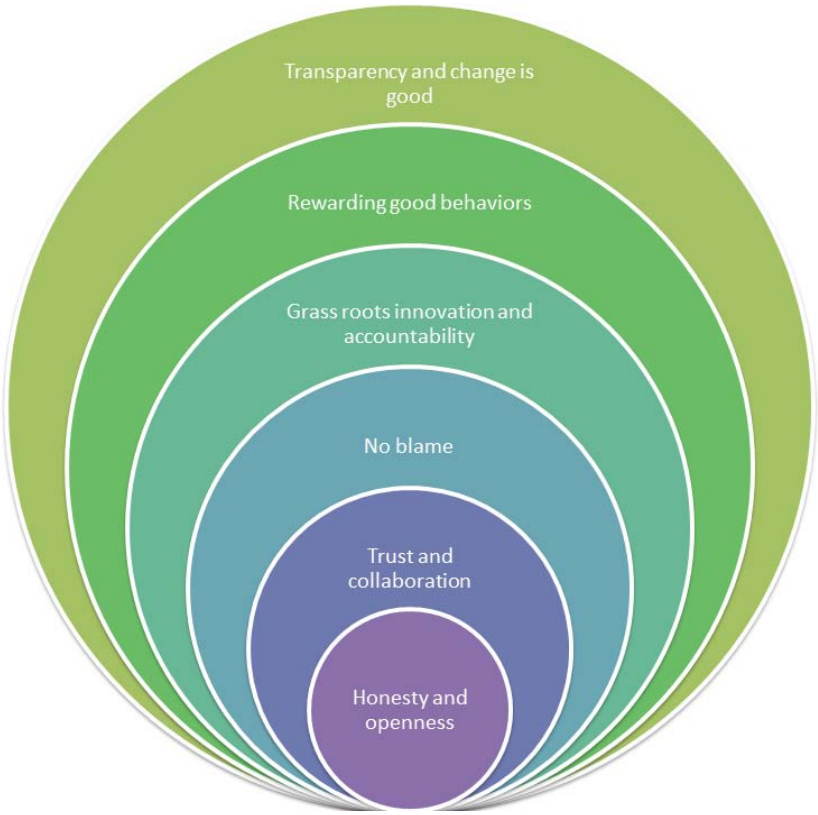
Summary

Just like the previous chapter, we have again covered quite a lot of ground in terms of the human side of implementing CD and DevOps. Let's spend a few moments summarizing what we have learned.

- Having an open and honest culture is essential for CD and DevOps to work as it allows for open and honest dialogue
- The working environment should also be as open as possible
- Trust is a powerful tool and allows for a greater degree of collaboration
- Playing the blame game does more harm than good and is not a way to motivate

- You should provide the opportunity and freedom for innovation but ensure individuals accept that with freedom comes accountability and responsibility
- Rewarding and incentivizing good behaviors and business milestones across the board is much better than focusing on release milestones or business processes
- Change is good and risk can be managed and reduced with CD and DevOps
- Be visible and transparent about everything you are doing

The following diagram provides a pretty good overview:



The many layers of CD and DevOps

We have now covered the tools, culture, and behaviors you'll need to successfully implement CD and DevOps. In the next chapter, we'll look at the hurdles you will encounter along the way and see what tricks and tips may help overcome them.

6
Hurdles to Look Out For

Up until this point in the book, we have been focusing on the core tools you'll need in your toolbox to successfully implement CD and DevOps. Along the way we've 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 look at ways that they can be overcome, 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 some 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 them and ensure that they don't ground you or completely run the implementation into 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 may 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. The sorts of hurdles you will encounter will include the likes of:

- 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 can help and/or hinder your progress

- A lack of external understanding or visibility of what you are trying to achieve may 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 non-technical)
- Recruitment

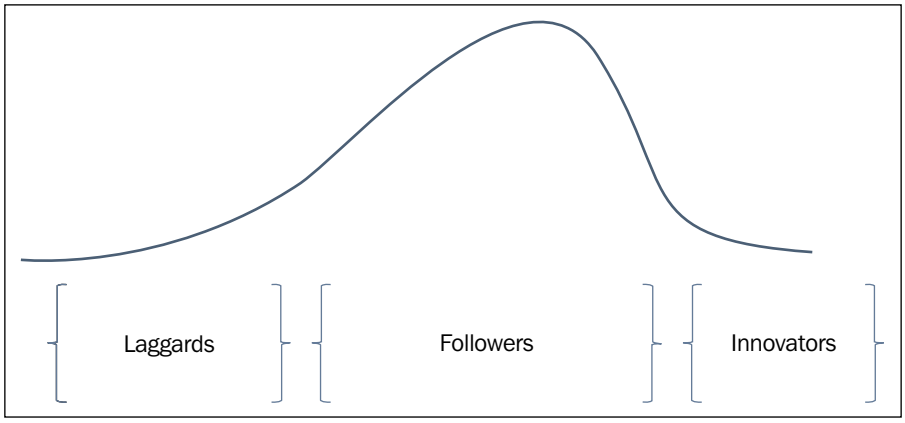
The list could be much longer but there's only enough space in this book so we'll focus on more obvious potential issues, which could, as previously mentioned, run the implementation of CD and DevOps into shallow waters or worse still onto the rocks. We'll start with focusing on individuals and how they 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 and your team 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 by all accounts 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 nth degree but what is important for you to realize is that one or two individuals who are loud enough can make a lot of unwanted noise and can redirect your attention from your vision and goal, which 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, and lastly the laggards who are undecided or not convinced it's the right direction to go. The following diagram illustrates these three types:



The three types of individuals identified during the early years of agile adoption

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 need some help to get over the crest so more attention is given to them. To focus on the laggards may 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 may 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 upon 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 may 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 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 trouble maker over a coffee—that 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).

Something that may increase the risk of dissenters spoiling the party is a lack of visible progress in terms of the CD and DevOps implementation. It may be that you're busy with some 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 may take this lull as a sign of the implementation faltering or they may even think that you're finished. As we covered in the previous chapter, being highly visible, even if there's not a vast amount going on, is very important. If people can see progress being made, they will continue to follow. If there is a period of perceived inaction then the followers may not know what way you are heading and may 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.

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

The change 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 should not be taken lightly and the impact on individuals should not be taken lightly; even if they originally thought it was the best thing since sliced bread.

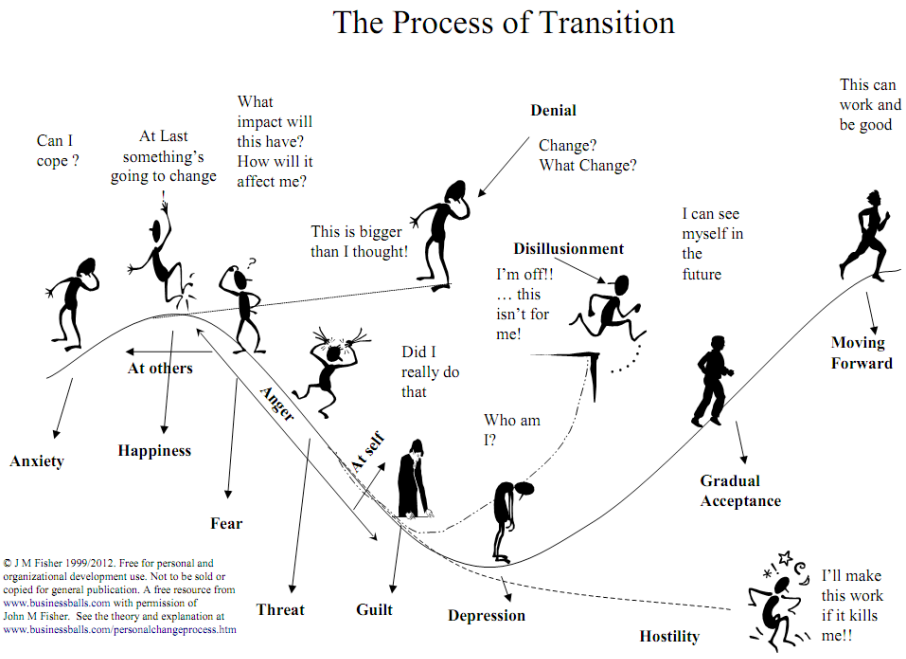
Those of you who have been in, or are currently in, management or leadership roles may well understand that change can be seen as both positive and negative, and sometimes it can be taken very personally—especially where a change to a business impacts individuals and their current roles within it. Let's 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 of the above. If we are mindful of these facts before we implement change, we have a clearer idea of what challenges to overcome during the implementation to ensure it is successful.

There has been much research into this subject and many papers 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 courtesy John Fisher from `http://www.businessballs.com/freepdfmaterials/processoftransitionJF2012.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:

- People may 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 that do not/cannot come out of the dip may 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 or 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 may 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 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 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 that light note we'll move onto the subject of what to do about those people within your organization who are not involved in your journey or may 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 an SME (small to medium enterprise) 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 not.

The following diagram illustrates how a typical corporate 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 may—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 on 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. If you experience wide reaching changes during your journey, and people are ignorant of what you're doing, your story may not end so well. Bear that 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 may seem like quite a lot of work for little gain but you may be surprised how much benefit it can bring. Say for example, you get the article written and published and it is read by the CEO or an SVP who then decides to visit and see what all the fuss is about. That is a major moral boost and good PR. Not only that but it may help with your management dissenters—if they see the high ups recognizing what you are doing as a positive thing they may 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 may have others who are well aware but are either restricted by or hiding behind corporate red tape. Let's spend some time looking into 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. If you work within the service sector and have commercial obligations to meet certain SLAs, or you may work within a financial institution and have regulatory guidelines to adhere to, you will be in some ways hampered in how you implement CD and DevOps. This as they say, comes with the territory. What you need to do is work with those setting and/or policing the rules to see what wriggle room you have. You may find that some of the rules and guidelines set in place for the business are actually overkill and have only been implemented as they have been because it was easier to stick to what it said in the book than it was to refine to fit the business needs.

The need for such rules, guidelines, and policies mainly revolves around change management and auditability. In simple terms, they offer a safety gate and a way to ascertain what has recently changed should problems occur. You may find that those managing or policing these rules, guidelines, and policies will consider CD and DevOps to be incompatible with their ways of working. This may be true but doesn't mean that it's correct.

During the investigation stage their organization/department may have been highlighted as an area of waste within the product delivery process (I would put money on it) so they may be defensive about change. It may even be the case that they simply don't know what they can change without breaking a rule or corporate policy. Work with these people and help them understand what CD and DevOps is about and help them research what parts of their process they can change to accommodate it. Do not simply ignore them and break the rules as this will catch up with you later down the road and could completely derail the process. Open, honest, and courageous dialogue is the key.

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

Geographically diverse teams

We previously touched on 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, the differences in culture. Not having a physical presence is always a barrier; however, there are a few things that you could look at to remove some of these as well:

- Ensure teams in your office have regular (ideally daily) teleconference calls with the remote team(s).
- 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.
- We all have access to some form of video conferencing—be that using a corporate (read expensive) teleconferencing system or something simple such 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 a virtual wall/window between the offices, which is always on so people on each side of the virtual wall/window can simply walk up and have face-to-face conversations.

- If your budget allows, try and get people swapped across the offices.
- Instead of relying on e-mail use a messenger or chat solution and encourage use of blogs, discussion boards, and forums.

It is of course going to be much more difficult to remove the physical barrier of many miles and sometimes many oceans—unless you have perfected the art of matter teleportation—so using some collaborative tools and approaches may well help. Another potential barrier is time zones, which can play havoc with things such as daily stand-ups (from experience these normally happen first thing in the morning), however, with some creative thinking you can overcome these small issues.

Previously I mentioned that differences in culture may be something to take into account. This should not be taken lightly. The reality is that in some parts of the world the culture may not be the fast and loose western culture where everyone has a voice and isn't afraid to use it. Instilling openness, honesty, and transparency may be more difficult for some and you should be mindful of this. Work with the local HR or management team, 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 the 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 may be situations that you didn't foresee or steps in the existing process, which were not picked up during the investigation stage. It might 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 may 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 and your team are working hard to instill a sense of openness and honesty so the worst thing you can do is the exact opposite.

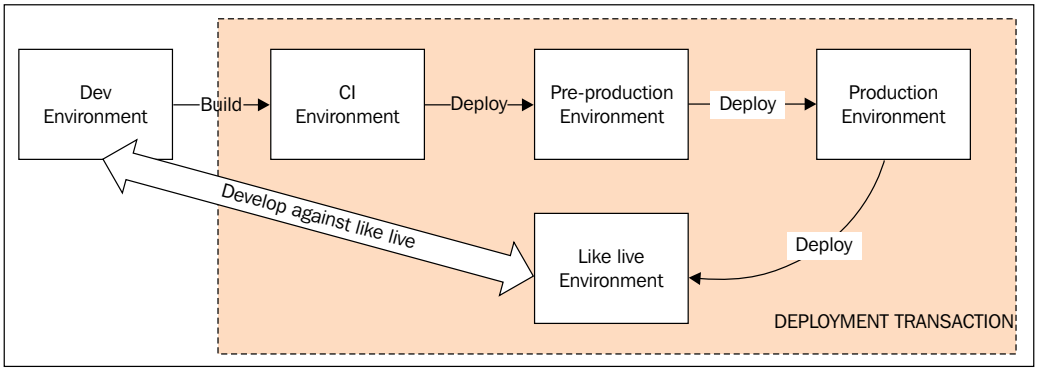
Admitting defeat, curling up in a fetal position, and lying in the corner whimpering is also not an option. As with any change things go wrong so review the situation, review your options, and move forward. Once you have a way to move around or even through the problem, communicate this. Ensure you're 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—a sort of lead by example. If you're using agile techniques such as scrum or Kanban to drive the CD and DevOps implementation, you should be able to change direction relatively quickly without impeding progress.

Okay so this is all very PMA (positive mental attitude) and may be seen by some of you who are more cynical than the rest as management hot air and platitudes, so let's look at another example.

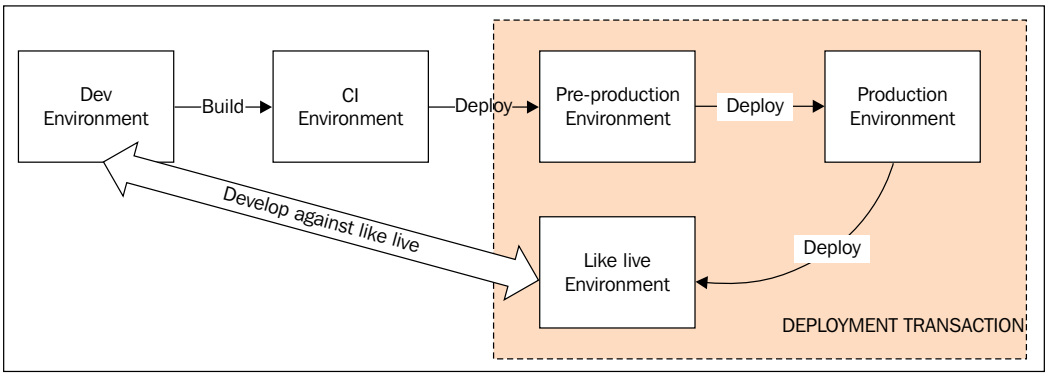
ACME systems implemented a deployment transaction model (covered in a previous chapter) to manage dependencies and ensure only one change went through to the production system at any one point in time. This worked well for a while but things started to slow down. Some open and honest discussions took place between the CD team and others within R&D and Operations as the slowdown was starting to hit the R&D team's ability to deliver. After much debate it turned out that the main source of the problem was that there was no sure way of determining which software asset 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 deployment transaction was starting to hinder CD.

The following diagram details the deployment transaction as originally implemented:



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 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 assets went to production. It was then down to the various R&D teams to manually work out in which order things should be deployed.

The following diagram depicts the revised deployment transaction boundary:



So ACME had a potential show-stopper, which 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 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 may 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; automated build of engineer's workstation, 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 not consistent 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.

Let's look at a very simple example; if you're a software engineer you will use an IDE to write code and you will use a compiler to generate the binary to deploy, and if you're a DBA you'll use a SQL admin program to manage your databases and write SQL. 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 the said tools again. It may 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 non-technical) you build and/or implement for your CD and DevOps adoption. You need to be confident that when you do the same actions over and over again, you 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 may be an infrastructure issue or you may have introduced a bug within the latest version of the CD tools).

All in all it may 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 and the efficiency grows and the organization starts to be recognized as one that can deliver quality products quickly (and it will), then growth and expansion may well become a high priority—which is great news. You now need to find individuals who will work in the *new way* and exhibit the behaviors you and your team have worked so hard to instill and embed throughout the organization. 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; there's not that many people out there with this specific trait nor is there much knowledge throughout the recruitment world of what *CD and DevOps* actually means.

You will therefore need to embark on more knowledge sharing with those involved in your recruitment process to ensure that they understand what you're looking for. You may need to do this number of times until this sinks in.

There are few things you can do to filter out those who get it and those who don't. One of my favorites is a very simple interview question:

> *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 may be onto a winner.

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

Summary

So what new things have we learned?

- There will be hurdles along the way; some you will see coming and some will take you by surprise
- Put your energies into those who wish to progress and the rest will follow
- Everyone is different and will react to what is going on around them in uniquely different ways
- Keeping those outside of the inner circle in the loop might stop decisions being made which could scupper your progress
- If things start to go wrong, take stock, look at the options, and move on
- Red tape can cause problems but there will be ways around them if you persevere
- Geography needn't be a problem if you look at simple ways to bring people closer
- Repetition is a good thing
- Hiring the right people may become trickier

There will no doubt be other hurdles, hazards, and potential blockers along the way but as long as you're prepared you will be successful. Talking of success, we'll now move onto the measurement of success and why it is so important.

7
Measuring Success and Remaining Successful

Over the previous chapters, we have looked at what tools and techniques you will need to successfully adopt CD and DevOps and highlighted some potential hurdles to overcome. With this information in hand, you should be in a good shape to succeed. In this closing chapter, we'll look at a couple of things those are as important as how you go about implementing CD and DevOps, those being how you measure success, and what you should do once you near your goal.

We'll start with the important but sometimes overlooked—or simply dismissed—area of monitoring and measuring progress. This, on the face of it may be seen as something that is only useful to the management types and won't add value to what you and your team are doing; however, there is no question that being able to demonstrate the progress made will add value. 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, so that we can plainly see and understand how far we have come and how far we have to go. To be able to do this effectively, we'll need to ensure we start looking into 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*.

One word of warning here, you may be tempted to implement something that will *do for now* as implementing effective monitoring and reporting solutions can be quite time consuming and may need some effort. Before you look at cutting corners, you should bear in mind that most of what you'll need to implement in terms of monitoring will actually be long lived and may well become an integral part of the new ways of working. If you don't deliver a quality solution from the start, trust in it will be eroded quite quickly. Just as we covered in the previous chapters, building trust is important. We'll start, as they say, at the beginning and focus 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 as you would think and there are many tools available to help you in this regard: code quality, code complexity, and unit test coverage analysis tools to name but three. You can also look at measuring things such as comments versus code or commit rates. However, you need to remember that these types of measurements can hide bad practice as well as highlight good. You just need to agree what is needed and what constitutes *good* and *bad* then investigate what tools are available that can determine and report when code is *good* or *bad*.

It sounds simple and to be honest it can be but you need to put some time and effort up front. It may be quite a bit of trial and error and tweaking as you go. It's all down to what is more important to you and how much trust the wider business has in your software quality and how much trust they have in the team(s) building and looking after the platform.

One thing to consider before you look at measuring software code metrics is how the software engineers will feel about this. Some may be guarded or defensive about this as they may see it as questioning their skill and craftsmanship in relation to creating quality code. You need to be careful you don't get barriers put up between you and the software engineering teams—there's going to be enough of them from other areas of the business as it is. You should sell these tools as a positive benefit for the engineers. For example: they have a way to prove how good their code is, they can use the tools to inspect areas of over complexity or areas of code that may be more at risk of being buggy, they can highlight redundant code and remove it from the codebase, they can visually see hard dependencies, which may help when looking at componentization, and so on. These types of tools are quite powerful and are bit like the Swiss army knife of software analysis (you can see this for yourself at `http://nemo.sonarsource.org/dashboard/index/org.codehaus.sonar:sonar?did=1`).

Another selling point for the vocal dissenters is that they can be involved in the setting up and configuration of the tool (for example, they could set the threshold at which code versus comments is set or what level of code coverage is acceptable). Whatever you do, you need to ensure you have the software engineers bought in to it.

Let's look in more detail at the various things you should consider as worthy of analysis.

Code versus comments

It is all well and good focusing on the code but 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 software engineers don't believe comments are worthwhile and believe that if another engineer cannot read the code, then they're not worth their salt. This may or may not be correct but including comments within one's source should be encouraged as good engineering practice and good manners.

One thing to look out for should you implement 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 may be a bit extreme but I'm sure if you look close enough through your codebase you may 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 upon outdated design patterns or even based upon an old language standard (for example, an older version of JAVA or C#) it may be quite time consuming trying to understand what the code is doing without at least some level of commenting.

The next thing that we'll look at is code complexity.

Code complexity

Having complex code in some circumstances is fine and sometimes is 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 or even a finance module, having overly complex code can do more harm than good. Complexity for complexity's sake is just showing off.

Having overly complex code can cause you lots of problems—especially when trying to debug or when you're trying to extend it to cater for 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 upon 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.

It is well worth putting some time aside to look into this in more detail before you dive into implementing tooling to provide this sort of information, as if you don't understand what the underlying principles are, it is going to be quite difficult to set up the tooling and agree on the correct thresholds.

The next thing to seriously consider is code coverage.

Code coverage

If you are not (for some extremely strange and unfathomable reason) incorporating unit tests within your software development process and/or not using test driven development, you are going to struggle to get anything meaningful out of analyzing unit test coverage. That isn't to say you can't develop quality code without including unit tests; if you believe this to be a false statement of fact then consider how much high quality software was developed prior to the time of TDD and test automation. If you are able to include unit tests, then there's no reason not to do so.

Incorporating unit tests within your software development process is always a good idea as this allows you to exercise code paths and logic at a much more granular and lower level and hopefully spot/eradicate bugs very early on. As you become more reliant on unit tests to spot problems early on, it's always a good idea to get some indication of how widespread the use is—hence the need to analyze coverage. For example, you may have only recently started to implement unit tests and want to ascertain which areas of your platform are covered and which areas not, so you can focus on some manual or integration tests.

If your software development process has unit tests as an integral part then having an idea of how much coverage you have is a good way of spotting potential gaps and areas of risk.

It should be pointed out that the metric being measured is broadly the percentage of the codebase that is covered by tests; therefore, if the business gets hung up on this value they may consider a low percentage 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. You should set the context and ensure everyone looking at the percentage figure understands what it means.

Let's now look at the effectiveness of measuring commit rates.

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 may block the progress of other engineers.

There may be a fear that if engineers commit too frequently, the chances of bugs being created increases, especially when you take into account the fact that there's an outside risk that unfinished code could be incorporated into the main code branch. To be honest this is not a real risk as no engineer would seriously consider doing such a thing—why would they? The real risk is if engineers hold off committing as there will be more code, which could cause more problems.

As we previously covered the best approach to delivering changes is little and often. This should not be restricted to software binaries; delivering small incremental chunks of 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. Some teams even use this kind of information to produce league tables (for example, the top 10 committers this week). However, this may be seen as a negative thing and may invite the wrong behaviors—use this method wisely.

Most modern source control solutions have built-in tools for reporting the usage and commit rates. If your chosen tooling doesn't come with such tooling there will be open source tools that you can use.

Some may also want to hook up their continuous integration solutions to be triggered at each commit. This does work well but can be a bit of a pain if you're trying to encourage frequent commits (if you have many engineers all working on the same codebase you may end up with a long queue of CI jobs waiting to start). You just need to use a bit of common sense.

Next we'll move onto unused or redundant code analysis.

Unused/redundant code

As your software platform matures, there will be quite a bit of refactoring of the codebase. New features added the odd bug fix, existing features were enhanced, old functionality deprecated, and so on. Over time this will mean that some of the source code will become redundant and unused. The tendency is to leave this code in situ as:

1. It may be needed again.
2. The software engineer is not too sure what will happen if the code is removed.
3. It takes effort to remove/unpick code.

It may be perfectly safe to leave redundant code within the codebase; however, it may cause more trouble later down the line. For example, if a bug is found and a software engineer is asked to look through the code to investigate the root cause and there is redundant code he/she may not be fully aware and may waste time scanning through lines of source code, which are to all intents and purposes dead. Worse still he/she may think they find the root cause in code, which is never executed.

It is therefore safer and good practice to remove any redundant code. Ascertaining what code should be removed can be a daunting and laborious task; however, there are some very good tools available to scan through your source code and highlight code that is no longer executed and is therefore a candidate for removal. It may be that you uncover a vast amount of redundant code, which can cause another problem—how to obtain the authority to spend many hours removing code that doesn't add business value. If this is the case then *sell* the drawbacks of leaving it there and get agreement to start small and build up.

We'll now look at duplicate code analysis.

Duplicate code

As with redundant code there is a high probability that your codebase may contain duplicated code and/or functionally, especially when looking at a large codebase. Sometimes this may be by design; sometimes it may be due to software engineers working on the platform that are not fully au fait with the codebase and create new functions/functionality, which already exists.

One good software engineering best practice is code reuse. So if you can utilize tools to highlight code duplication, then you also have candidates for turning the said code into a share library or some such. Having duplication in the codebase is not necessarily a risk, but it can lead to more work should you need to change the same piece of code scattered across multiple areas of your platform.

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

Adherence to coding rules and standards

You may well have coding standards within your software development teams and/or try to adhere to 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.

This type of analysis will take some setting up as it is normally based upon 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 the tooling.

This all sounds like quite a bit of work so is it actually worth it? Yes!

Where to start and why bother?

All in all there are many things that you can and should be measuring, analyzing, and producing metrics for. You just need to work out what is most important and start from there. Some of the tools available are extremely powerful and can give you most of what you need out of the box. You will, however, need to put some effort into getting the tools set up correctly. This is a great opportunity to bring into play some of the good behaviors you want to embed: collaboration, open honest dialogue, and so on.

Implementing these sorts of tools may seem like you're implementing a stick to beat engineers with but it should really be seen as (and sold as) an extremely powerful way for them to see where the problems lie early on so that they can put some focus into addressing the areas of potential risk before they become live issues.

As previously stated, it is beneficial 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 may be some questions around the validity of doing this when it doesn't directly drive the adoption forward. It may not directly affect the adoption but it offers a few very worthwhile additions:

- 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

One thing you may also want to consider is including this kind of measurement and tooling within the CI process. That way you have some additional small and discreet quality gates to ensure the code is as it should be. For example, let's assume you have the CI job for a given software component set up to run the build and automated testing process; if these steps are successful the software can be released. If you then add checks for code quality and the analysis comes back with an unfavorable result, the software component goes no further. You then have a way to prove that the software is written well, passes all of its tests, and is therefore worthy of releasing.

One caveat here is that most of the tools available focus on mainstream software development languages (Java, C#, and so on) so there may be some restrictions within your tool selection. That said there's no reason you shouldn't look around and/or build your own.

We'll now move our focus from the software and look at the importance of having stable and consistent environments available to use throughout the product delivery process.

Measuring the real world

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

Measuring stability of the environments

As previously covered, you will most probably have a number of different environments which 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 to 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:

- Why did your overnight tests fail? Looks like a network or database blip, so must be an environmental issue. The build server has just dropped off the network. Must be an environmental issue.
- I can't seem to commit my changes. Must be an environmental issue.
- Why can't users log in? I know my code is OK so it must be an environmental issue.
- Why is the live platform not working? It must be an environmental issue.

We've all heard this and some of us will be guilty of saying this kind of thing ourselves. All in all it's not helpful and to be honest may 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 we were to group all of these tests together and run them continuously against a given environment? We would end up with most of the platform software (the degree of coverage depends on the number of tests you have) being tested over and over again. If we capture the output of this to a simple dashboard, which provides an overview of the results (for example, *green block* = pass and *red block* = fail), we can quickly and easily see how healthy the environment is or more precisely we can 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 may have different tests written in different ways using different technologies, which may not play well together
- Some tests may well conflict with each other, especially if they rely on certain pre-determined sets of test data being available
- The tests themselves may not be bullet proof and may not show issues, especially when they have mocking or stubbing included
- Some of your tests may *flap*, which is to say they may not be consistent and may fail now and again for no reason
- It may take many hours to run all of the tests end to end (on the assumption that you are running these sequentially)

Assuming you are happy to live with the caveats or you have resource 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 and therefore you should be able to spot instability issues within a given environment with relative ease; sort of.

Combining automated tests and system monitoring

Just running tests only gives you half the story. To get a true all round picture we should also continuously monitor the infrastructure on which the software is running to ensure that it is behaving as expected. The sort of things we can measure should be the standard output from many of the infrastructure monitoring solutions available (for example, CPU utilization, storage space, memory utilization, network throughput and latency, and so on).

If we take the output from both solutions and display the results side by side, we start to see how stable the environment as a whole actually is and more importantly we start to get an indication of where problems originate from. OK so I've made this sound quite simple and to be honest the overall objective is simple; the implementation may be somewhat more difficult.

If you want to fully combine the outputs into a single uniformed dashboard there's going to be some effort required to ensure the outputs from both the automated tests and the infrastructure monitoring can be represented in the same format and combined into a single dashboard solution. There are a few solutions available (open source and commercial) but there's nothing stopping you creating some sort of bespoke solution if you have the time and available resource.

Ultimately what you want to be able to do is ensure that the entire environment is healthy—I'm classing the environment here as the infrastructure and software combined. This way if someone says *it must be an environmental issue*, they will actually be correct.

If we pull all of this together we can now expand on the above 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 have thus far been focusing on how to prove that the platform as a whole is stable and functions in the way you expect, using your automated tests and monitoring. This will help give an indication of how things are functioning but may not provide enough detailed information to be able to ascertain if a given environment is truly healthy.

Added to this is the fact that this method will be restricted to non-production environments; you will have issues trying to run automated tests in the production environment (unless your operations team are happy with test data being generated and torn down within the production database many times per hour/day). What you need to therefore look at is some more in-depth real-time monitoring and metrics.

Real-time monitoring of the software itself

We have already covered how you can measure or prove the stability of the various environments you rely on by using system monitoring and running automated tests. This will give you some useful data but this only proves one thing: the platform is up and the tests pass. It does not give you an in-depth understanding of how the platform is behaving, or more importantly how it is behaving in the production environment being used my many real-world users. To achieve this you need to go to the next level.

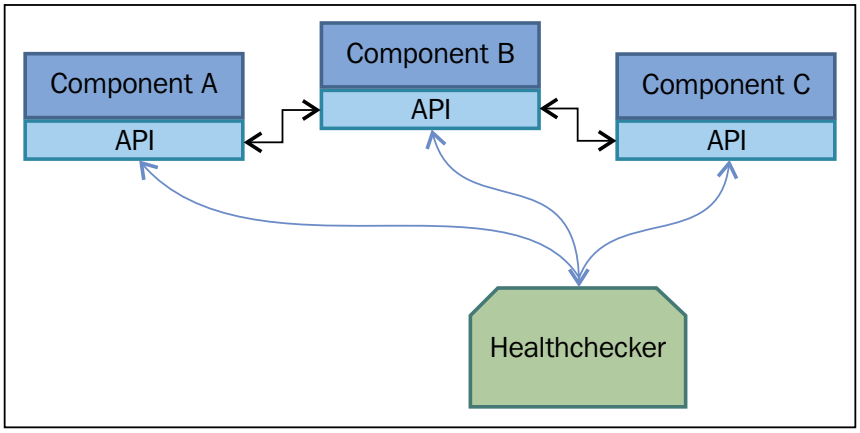
Consider how a formula one car is developed. We have a test driver sitting in the cockpit who is generating input to make the car move; their foot is on the accelerator making the car move and they are steering the car to make it go around corners. You have a fleet of technicians and engineers observing how fast the car goes and they can see data about 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.

The same goes for a software platform. 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 few solutions available that allow you to trawl through such outputs and compile them into useful and meaningful reports and graphs that you could combine with your monitoring and test results to get a more rounded view of the stability and health of the platform. There is a *but* here; it's very difficult to make this real time, especially when there's a vast amount of log data being produced. There's also the question of storage space as these types of logs can be very verbose and get very large very quickly.

A cleaner approach would be to build something into the software itself, which can produce this type of data for you in a small, concise, and consistent format. Let's assume that your software components contain APIs, which are used for component-to-component communication. If you were able to extend these APIs to include some sort of health check functionality, you could construct a tool which 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 may 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 which *calls* each component via the APIs and gets back data which 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 if 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 does rely 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 *number of open database connections*, you will need to change the APIs and redeploy all of the components and then update the tooling to accept this new data element. 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; that being the component itself generates the metrics you need and pushing the data to your tooling. Something like graphite (`http://graphite.wikidot.com/`) does this very well. Instead of extending the APIs you simply implement a small amount of code, which 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.

One additional advantage of going down this route is that you could also post data from your CD tooling to indicate when a release happened. This can be quite a powerful and useful way of spotting release related incidence almost immediately. For example, if you're monitoring the TPS (transactions per second) of a given software component and notice that it suddenly drops just after a release, there's a good chance that you've released a bug. You could then roll the change back (that is, re-release the previous version) and see if the TPS goes back to as it was.

Whatever solution you look at implementing (be it tooling to *call* each component and *pull* data or something like graphite, which allows each component to *push* data to it, or even a combination of the both) you will end up with some very rich and in-depth information. If you overlay this with the automated testing and system monitoring output, you pretty much have as much data as your average formula one technician (that is, lots of data). You just need to pull it all together into a coherent and easy-to-understand form. This challenge is another one for encouraging DevOps behaviors as the sort of data you want to capture/present is best fleshed out and agreed between the engineers on both sides.

We'll now move on from the in-depth technical side of measuring progress and success and look at the business focused end.

Measuring effectiveness of CD

Implementing CD and DevOps is not cheap. There's quite a lot of effort required, which will translate 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 technically minded individuals but your average middle manager may not get the subtleties of what it means and to be honest you can't really blame them. Seeing huge amounts of data and charts containing information such as TPS counts for a given software component or how may 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 which are important is the improvements in efficiency and throughput, as these translate directly into how quickly products can be delivered to 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 which 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 working software being in production
- A count of the release candidates that have been built
- A league table of software components which 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 following:

	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. You could also consider including some sort of league table (for example, which engineer released the most in the week), which can help instill a sense of friendly competition, as long as it's friendly.

What would also be valuable, especially to management types, is financial data and information, such as 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 may help provide focus for the engineering teams as they will start to understand how much these things cost. You could make this relatively simple and use something like an average hourly rate for an engineer and then add more details as they become available.

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

You are striving to encourage openness and honesty throughout the organization, therefore sharing all of the metrics and data you collect during the implementation will provide a high degree of transparency. You may well get some resistance in terms of *airing dirty laundry in public* but this is simply an excuse. If individuals are afraid of sharing bad news (maybe the quality of the software is below the level previously and rigorously broadcast by a software development manager or parts of the infrastructure are indeed pants) then that's just too bad they just need to get over it.

At the end of the day, every part of any business turns into spreadsheets and graphs (financial figures, head count, public opinion of your product) so why should the product delivery process be any different? If you are capturing quality data and metrics and sharing this you're one step ahead and when questions are asked you have the facts and figures in everyone's field of vision.

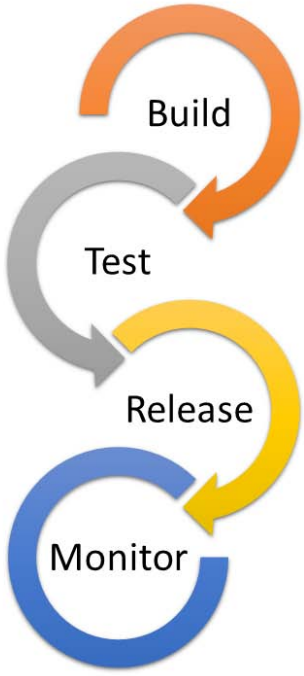
We'll now move on from measurements and statistics and look into the future (sort of).

Inspect, adapt, and drive forward

We've covered quite a lot over the last few chapters and I hope you found it useful, enlightening, and not too scary. You will hopefully have picked up some tricks and tips and now have enough background information to understand the following:

- The sort of things you will need to put into place to realize the benefits of CD and DevOps
- The tools and processes you can use to identify and highlight problems that are hiding in plain sight
- The sort of hurdles (both technical and non-technical) you will encounter and need to overcome
- How each person will react in a unique and different way to the changes you will be implementing
- The importance of culture, behaviors, and environment and the impact, positive and negative, they can have
- The advantages of having a dedicated and focused team responsible for the implementation of CD and DevOps
- How important it is to broadcast and share information and data with as many people as possible
- Why good PR is valuable

As we enter the closing stages of the book, we'll presume that at this point you are actively implementing CD and DevOps within your organization and in fact have been doing so for some time. You set out your goal and vision, you have a dedicated team around you, and you have been hard at work to implement the goal. The business has started to see the benefits and reap the rewards in terms of being able to deliver quality features to the market sooner. You have reduced the process of delivering software to something as simple as this:



A nice simple process for delivering software

It has been said already and I will say it again—you should never forget your original goal and vision. Along the way things may (will) sidetrack you, meaning that you and your team may well have to refine the direction a number of times but if you stick to your guns and keep moving in the direction which you set out on things will come good. It's not going to be quick, easy, or painless but you will succeed and you will reap the rewards. It will take time and effort to reach your goal and it will be worth it.

As previously stated, we'll presume at this point in the book that you are nearing the end of your journey. You've come a long way and there's not much further to go. The end goal is in sight and you are getting ready for the last 100 yard sprint. Before you do this you need to be aware that things will not be as clean cut here either. There are still a few hidden gems to keep you busy a little while longer.

Are we there yet?

Let's take stock of how things are as you near your goal. 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 but—and it's very important but that's not the end of it.

You will no doubt start to hear comments such as *we can deploy quickly so we must be finished* or *I've seen developers and operations people working together so we must have implemented DevOps* and questions such as *how much longer does this special little project need to run for?* This is normal for any project and is to be expected, especially from those who don't fully understand what you are doing or what you have achieved. The thing to remember and realize is that what you envisioned as the major issues at the beginning of the project will soon be a thing of the past and will be replaced by new and just as interesting issues. To explain we have to go off on a bit of a tangent.

Streaming

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

- In the very beginning many small streams flow downhill to converge into a river. This river flows along but the progress is impeded by a series of locks and a massive dam.
- The river then backs up and starts to form a reservoir.
- Every few months the sluice gates are opened and the water flows freely but it is normally a short-lived and frantic rush.
- As you identify and start to remove the manmade obstacles, the flow starts to become more even but it was hindered by some very large boulders further down stream.
- You then set about systematically removing these boulders one by one, which again increased the flow, which in turn starts to become consistent, predictable, and manageable.

- As a consequence of removing the obstacles, the water level starts to drop and small pebbles start to appear and create eddies, which are restricting the flow to a small degree but not enough to halt it.
- The flow carries on increasing, the water level 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.

So what's this got to do with implementing 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 are the streams into the river 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 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 manmade locks and dams, the things you knew about when you started out. As you remove these you will get some traction and will start to see some positive and interesting results. You start to see the hurdles (boulders), address them and the results, and efficiencies start to become obvious for all to see. As this goes on, those involved will start to forget about the original pains and problems and start focusing on new pains and problems, which were not evident—or rather were miniscule in comparison—when you started out. These are the new boulders. They are simple things such as the following:

- Why is it taking 10 minutes to complete a build and automated test run? Surely we can do that in seconds.
- Why do we need to spend 5 minutes filling out release documentation? Surely we can automate that.
- Why do we need to release software components in sequence? Surely we can parallelize this.

- Why do MySQL database updates take so long? Surely we can streamline this or look at other storage platforms.
- Why do we have to pre-build VM servers? Surely we can provision on the fly.
- Why do we need to wait for days to get network changes made? Surely we can implement some sort of IaaS.

In the space of a few months, the vast majority of those 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 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 something positive. The teams were constrained and unable to truly innovate, or get out of the mire, due to the complexity of the big release process. They no longer have to worry about that as the process of releasing software has become everyday background noise and it just happens over and over again without much effort required. As new people come on board they will have no knowledge or experience of the dark days and they take for granted that software can be released with ease.

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. Now they are something real and they need to be addressed, otherwise things may start to slow down and the days will again become darker.

Does this mean that your original goal has not been met and you have failed? No it doesn't. It just means that the goal may need to be tweaked and refined, inspected and adapted. The landscape has changed therefore so does the plan. Does this mean that you and your team have to start all over again? No it does not. However, it does mean that now is a good time to work out an exit strategy for you and your team.

Exit stage left

The business has gotten used to the changes you and your team spent many many hours implementing and it is now looking at new issues. The question is who should address these new found challenges? The answer is quite simple: not you and your team. You have embedded the new collaborative ways of working, helped bridge the gap between Dev and Ops, helped to implement new tools and optimized processes, drank lots of coffee, and had little sleep. It's now time for those you have helped to step up.

The wider business has the tools, knowledge, confidence, and experience so as you near your 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 will have reached or be very close to reaching your goal, be that *to release fully working quality software to production 10 times per day* or *remove the barriers between Dev and Ops* so as with any good agile ways of working, it's time to inspect and adapt.

Your focus should now change from delivering to assisting the delivery. Your team should now start pushing those within their network 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 as you will have been through much worse together.

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. We covered how to use retrospection and other tools to look back and plan forward, how open, honest, and courageous dialogue helped to find the correct path. Now think of the boulders in the water as elephants 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 elephants very quickly and now has the experience, confidence, and expertise to remove them quickly and painlessly. That said you should ensure that you do not become complacent.

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 a good thing and you and your team should be very proud of what you have achieved. That is no reason to rest on your laurels; it may be tempting but now is not the time to simply sit back and admire your handy work. You have helped the business to evolve but you have to be very mindful of the fact that the business can start to devolve as easy as it can continue to evolve 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 may start to become noisy again and the followers may start to listen to them.

You and your team will have shifted your position from *doers* to *facilitators* and *influencers*, so you need to ensure you all keep this up. Make sure you are still visible and there to help and assist where needed. Just like good parents you have set up a safe environment for growth and self-discover and therefore 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 may seem simple but it can be much harder at times; you're used to doing stuff yourselves and now have to help and watch others doing stuff. It's sometimes harder but just as rewarding. You and your team have now taken the next step in your 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.

Wider vision

CD and DevOps should not be restricted to simply software/product delivery. The tools, process, and best practices that come with this way of working can be extended to include other areas of the business. 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, which 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 previously covered, 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 better, 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 pre and post stages

Doing this may well 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.

This is of course merely a suggestion; it all depends on your business, the business needs, and where the big pain points are. Whatever you do or don't do in terms of widening your remit, you will find that you and your team have moved beyond hands-on implementation of CD and DevOps solutions and you now have some well-earned time on your hands.

What's next?

So you and your team have delivered what you set out to do and things are working well, better than you envisaged even. The business has all grown up, can tie its own shoe laces, and doesn't need you anymore—well not quite. If you consider where most of the individuals within the business started and where they have ended up, you will most probably find that they are at the same evolutionary point you and your team were when you started out on this adventure. You and your team have evolved further and you have become the holders of knowledge and experience. You are the subject-matter experts. You know your stuff.

Depending on how big and diverse your business is there will always be some who need assistance, help, and guidance. The CD and DevOps landscape is always changing and growing: new ways to do things, new tools, new ideas, new insights. Just trying to keep up could take most of your time and attention. You may well have hooked yourselves into the wide CD and DevOps communities, so you can share your findings and experiences with others and bring others' experience and knowledge back into your business.

It may make sense for you to retain a dedicated team to further advance the CD toolset, train up new people, evangelize, and continue to keep an eye on things to ensure nothing slips back to the bad old days. Or it may make more sense to disband and embed the team back into the engineering community from whence they came to keep things moving at grass roots.

Maybe the very act of implementing CD and DevOps vastly increased the viability of your business so that the growth accelerates quickly, say through acquisition, and you need to widen the adoption. In the end it boils down to the way your business works—every business is different—however, you need to ensure you don't end up with elitists who become disconnected from the day to day. It's your call.

Summary

This book, like all good things, has come to an end. As has been previously pointed out numerous times, 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 upon experience and observations.

Even if you are simply window shopping and looking at what is needed to implement CD and DevOps, you should now be aware of what you are letting yourself and your organization in for; forewarned is forearmed and all that. It's not all plain sailing; there will be some interesting challenges which you will encounter along the way and will need to overcome them or rather you will overcome them.

The main things to remember are:

- CD and DevOps is not just about technical choices and tools; a vast amount of the success is built on the behaviors, culture, and environment.
- 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 (for example, you will struggle to have Dev teams and Ops teams working closely together if you can't release quickly and releasing quickly needs Dev and Ops teams to work closely together).
- Where you do need to make technical choices, such as tooling, look around or ask for advice. Don't settle on something that is *pretty close* and have to change the ways of working to fit the tooling.
- If bespoke in-house software solutions and tools fit better, try and build them. Maybe even open source them when you've finished to help others who are starting out.
- 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 who 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 for implementing 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 from there—just as you would with any good agile methodology.
- As is the agile way ensure you have *inspect and adapt* deeply embedded into the psyche of everyone involved and when you hit hurdles bring this to play to overcome or work around them.
- Share failures and successes so that you learn and others have the opportunity to learn from you.
- Above all have fun and—this is very important—don't under any circumstances give up.

Good luck!

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

V

Thank you for buying

Continuous Delivery and DevOps: A Quickstart Guide

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, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike.
For more information, please visit our website: `www.packtpub.com`.

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, 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.

[PACKT] PUBLISHING

Applied Architecture Patterns on the Microsoft Platform

ISBN: 978-1-84968-054-7 Paperback: 544 pages

An in-depth scenario-driven approach to architecting systems using Microsoft technologies

1. Provides an architectural methodology for choosing Microsoft application platform technologies to meet the requirements of your solution
2. Examines new technologies such as Windows Server AppFabric, StreamInsight, and Windows Azure Platform and provides examples of how they can be used in real-world solutions
3. Considers solutions for messaging, workflow, data processing, and performance scenarios

---

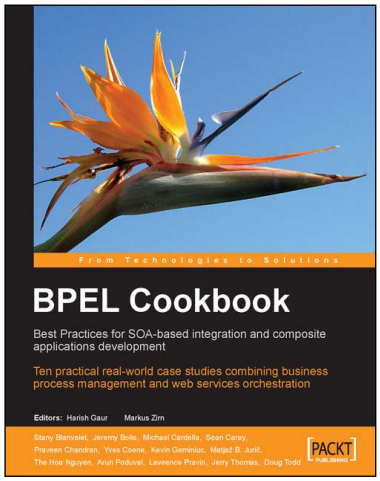
Open Text Metastorm ProVision® 6.2 Strategy Implementation

ISBN: 978-1-84968-252-7 Paperback: 260 pages

Create and implement a successful business strategy for improved performance throughout the whole enterprise

1. Fully understand the key benefits of implementing a business strategy
2. Utilize features like the integrated repository and ProVision® frameworks
3. Obtain real insights from practitioners in the field on the best strategic approaches
4. Ultimately design a successful strategy for deploying ProVision®

[PACKT] PUBLISHING



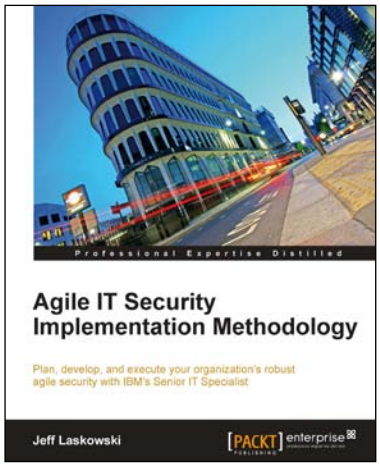
BPEL Cookbook: Best Practices for SOA-based integration and composite applications development

ISBN: 978-1-90481-133-6 Paperback: 188 pages

Ten practical real-world case studies combining business process management and web services orchestration

1. Real-world BPEL recipes for SOA integration and Composite Application development
2. Combining business process management and web services orchestration
3. Techniques and best practices with downloadable code samples from ten real-world case studies

---



Agile IT Security Implementation Methodology

ISBN: 978-1-84968-570-2 Paperback: 120 pages

Plan, develop, and execute your organization's robust agile security with IBM's Senior IT Specialist

1. Combine the Agile software development best practices with IT security practices to produce incredible results and minimize costs
2. Plan effective Agile IT security using mind mapping techniques
3. Create an Agile blueprint and build a threat model for high value asset