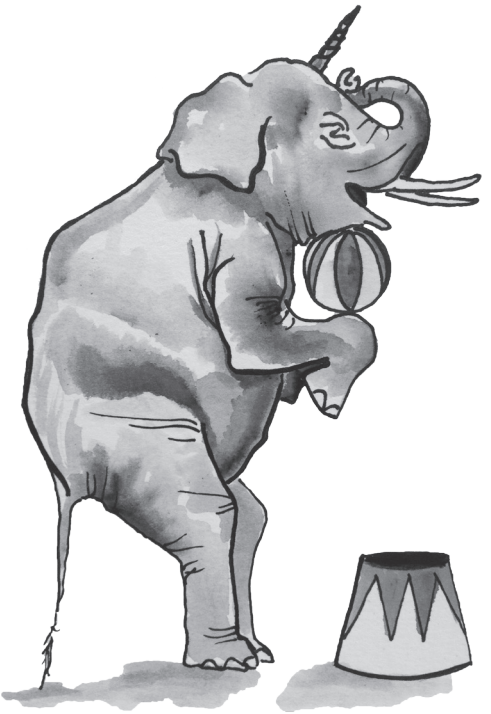
Starting and Scaling DevOps in the Enterprise

Gary Gruver



“With all of the hype around DevOps, it has been difficult to really understand how DevOps scales at large companies and how to begin. The insight in this book has provided clarity and a path forward. I look Forward to beginning the process using this book as the guide.”

- Steven D. Leist, VP, American Airlines

Print ISBN: 978-1-48358-358-7

eBook ISBN: 978-1-48358-359-4

Elephant pictures by Amelia Tiedemann

Graphics by Shahla Mahdavi and Cassie Lydon of Katie Bush design

TABLE OF CONTENTS

ABOUT THE AUTHOR

Gary Gruver is an experienced executive with a proven track record of transforming software development and delivery processes in large organizations, first as the R&D director of the LaserJet firmware group that completely transformed how they developed embedded firmware and then as VP of QA, Release, and Operations at Macy's.com where he led the journey toward continuous delivery. He now consults with large organizations and runs workshops to help them transform their software development and delivery processes. He is the co-author of *Leading the Transformation: Applying Agile and DevOps Principles at Scale* and *A Practical Approach to Large-Scale Agile Development: How HP Transformed LaserJet FutureSmart Firmware*.

Website: GaryGruver.com

Twitter: @GRUVERGary

Linkedin: https://www.linkedin.com/in/garygruver

Email: gary@garygruver.com

ACKNOWLEDGMENTS

Many people have contributed to this book. I would like to thank everyone I have worked with over the years who helped me better understand how to develop software. The ideas shared in this book are an accumulation of everything I have learned from working with each of you on a constant journey of improving software development processes. Without these discussions and debates, my understanding would not be as rich and the book would not be as complete.

I would like to especially thank all the clients of the executive and execution workshops for letting me join you on your journey. Sharing the challenges you were facing and the improvements that worked helped to fine tune the content of this book. Thanks also to Paul Remeis and Greg Lonnon for helping to fine tune the content by helping me deliver and improve the execution workshops.

I would like to thank everyone that has taken time to give me feedback on early versions of the book (in alphabetical order): John Ediger, Mirco Hering, Jez Humble, Tommy Mouser, and Vinod Peris. Your input significantly improved the final product.

I would also like to thank the editorial and production staff: Kate Sage, the editor, did a great job of forcing me to clarify the ideas so they could be communicated clearly and concisely. The back and forth made for a better book, but more importantly it required me to crisp up the concepts, enabling me to be more efficient at helping others on their journeys. Shahla Mahdavi and Cassie Lydon from Katie Bush design provided most of the graphics. They did a great job if creating visual artifacts to help communicate the ideas I am trying to get across. Finally, I would like to thank Amelia Tiedemann for the wonderful elephant pictures and cover design. I feel she was really helpful in communicating that a successful DevOps transformation requires more than just having all the right parts.

FORWARD

When David Farley and I wrote the Continuous Delivery book, we thought we were tackling a dusty, niche corner of the software delivery lifecycle. We didn't expect a huge amount of interest in the book, but we were sick of seeing people spending weeks getting builds deployed into testing environments, performing largely manual regression testing that took weeks or months, and spending their nights and weekends getting releases out of the door, often accompanied by long outages. We knew that much of the software delivery process was hugely inefficient, and produced poor outcomes in terms of the quality and stability of the systems produced. We could also see from our work in large enterprises that the tools and practices existed that would remove many of these problems, if only teams would implement them systematically.

Fortunately, we weren't the only ones who saw this. Many others—including Gary—had come to the same conclusion across the world, and the DevOps movement was born. This movement has had unprecedented success, primarily because these ideas work. As I've worked with leaders in large, regulated companies, and most recently as a US federal government employee at 18F, I've seen order of magnitude improvements in delivery lead times accompanied by improvements in quality and resilience, even when working with complex, legacy systems.

Most important of all, I've seen these ideas lead to happier technology workers and end users. Using continuous delivery, we can build products whose success derives from a collaborative, experimental approach to product development. Everybody in the team contributes to discovering how to produce the best user and organizational outcomes. End users benefit enormously when we can work with them from early on in the delivery process and iterate rapidly,

changing the design of systems in response to their feedback, and delivering the most important features from early on in the product lifecycle.

Gary has been applying ideas from the Continuous Delivery and DevOps playbook from well before these terms became popular, starting with his work at HP leading the FutureSmart LaserJet Firmware team. His large, distributed team applied continuous delivery to printer firmware, and showed the transformational results this created in terms of quality and productivity in a domain where nobody cared about frequent deployments. Then he went on to do the same thing in a regulated organization with complex, tightly coupled legacy systems.

Today's technology leaders understand the urgency of transforming their organizations to achieve both better quality and higher productivity. Effective leadership is essential if these kinds of transformation are to succeed. However overcoming the combined obstacles of organizational inertia, silo-based thinking and high levels of architectural complexity can seem like an overwhelming task. This book provides a concise yet thorough guide to the engineering practices and architectural change that is critical to achieving these breakthrough results, from a leader's perspective.

This book won't make your journey easy—but it will serve as an invaluable map to guide your path. Happy travels!

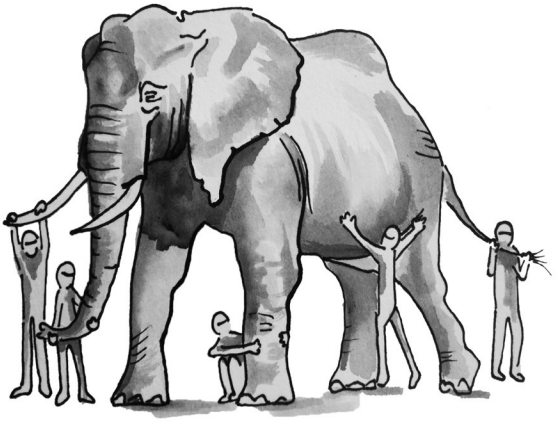
Jez Humble

Chapter 1

DEVOPS AND THE DEPLOYMENT PIPELINE

Software is starting to play a much larger role in how companies compete across a broad range of industries. As the basis of competition shifts to software, large traditional organizations are finding that their current approaches to managing software are limiting their ability to respond as quickly as the business requires. DevOps is a fundamental shift in how leading edge companies are starting to manage their software and IT work. It is driven by the need for businesses to move more quickly and the realization that large software organizations are applying these DevOps principles to develop new software faster than anyone ever thought possible. Everyone is talking about DevOps.

In my role, I get to meet lots of different companies, and I realized quickly that DevOps means different things to different people. They all want to do "DevOps" because of all the benefits they are hearing about, but they are not sure exactly what DevOps is, where to start, or how to drive improvements over time. They are hearing a lot of different great ideas about DevOps, but they struggle to get everyone to agree on a common definition and what changes they should make. It is like five blind men describing an elephant. In large organizations, this lack of alignment on DevOps improvements impedes progress and leads to a lack of focus. This book is intended to help structure and align those improvements by providing a framework that large organizations and their executives can use to understand the DevOps principles in the context of their current development processes and to gain alignment across the organization for successful implementations.



Part of the issue with implementing DevOps principles and practices is that there are so many ideas out there about what DevOps is, and so many different ways to define it. The most consistent and comprehensive definition I have heard lately is from Gene Kim, a co-author of The Phoenix Project and The DevOps Handbook. He is a great thought leader and evangelist for the DevOps movement. In order to get us all on the same page for our work here, we will use his definition of DevOps:

> DevOps should be defined by the outcomes. It is those sets of cultural norms and technology practices that enable the fast flow of planned work from, among other things, development through tests into operations, while preserving world class reliability, operation, and security. DevOps is not about what you do, but what your outcomes are. So many things that we associate with DevOps, such as communication and culture, fit underneath this very broad umbrella of beliefs and practices.

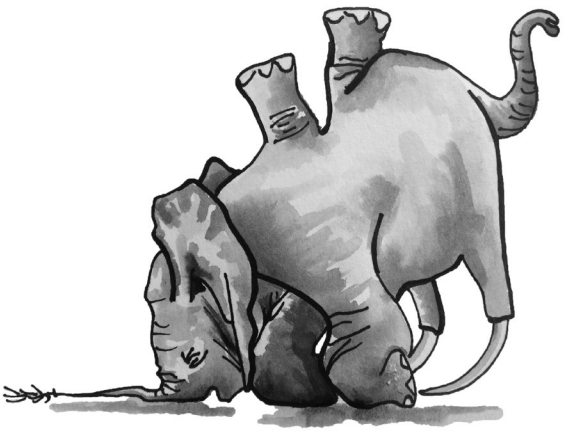
People have such different views of DevOps because what it takes to improve quality and flow at every step, from a business idea all the way out to working code in the customer's hands, differs for different organizations. The DevOps principles designed to improve this process are a lot about implementing changes that help coordinate the work across teams. The movement started with leading edge, fairly

small companies that were delivering code more frequently than anyone thought possible. DevOps was also very successful in large organizations like Amazon where they re-architected their monolithic system to enable small teams to work independently. More recently, DevOps has started being leveraged into large organizations with tightly coupled architectures that require coordinating the work across hundreds of people. As it started scaling into these larger more complex organizations, the problem was that people started assuming the approaches for successfully coordinating the work across small teams would be the same and work as well for coordinating the work across large organizations. The reality is that while the principles are the same for small and complex, the implementations can and should be different.

Most large organizations don't have that context as they start their DevOps journey. They have different people in different roles who have gone to different conferences to learn about DevOps from presentations by companies with different levels of complexity and different problems and have come back with different views of what DevOps means for them, like when the five blind men describe the elephant. Each stakeholder gives a very accurate description of their section of the DevOps elephant, but the listener never gets a very good macro view of DevOps. So, when they go to create their own elephant, nobody can agree on where to start, and they frequently want to implement ideas that worked well for small teams, but are not designed for complex organizations that require coordinating the work of hundreds of people. The intent of this book is to provide the overall view of the elephant to help large organizations gain a common understanding of the concepts and provide a framework they can use to align the organization on where to start and how to improve their software development processes over time.

This is important because if you can't get people in a large organization aligned on both what they are going to build and what approach they are going to use for prioritizing improvement, they are not very likely to deliver a DevOps implementation that will deliver the expected results. It will potentially have pieces of the different things

that the organization has heard about DevOps, but it won't really help the organization deliver code on a more frequent basis while improving or maintaining all aspects of quality. It is like having the five blind men build an elephant based on their understanding of the animal. It may have all the right parts, but it doesn't really look like or work like an elephant because they don't have a good macro view of the animal.



To clarify the macro view of DevOps, we will look at how a business idea moves to development, where a developer writes code, through the creation of the environment to how code gets deployed, tested, and passed into production where it is monitored. The process of moving from a business idea all the way out to the customer using a deployment pipeline (DP) was originally documented by Jez Humble and David Farley in their book Continuous Delivery. This book will leverage that framework extensively because I believe it represents the basic construct of DevOps. It captures the flow of business ideas to the customer and the quality gates that are required to maintain or improve quality.

It is my personal experience that creating, documenting, automating, and optimizing DPs in large software/IT organizations is key to improving their efficiency and effectiveness. You already have in place something that you are using to get code through your organization from idea to production, which is your DP. But documenting

that so everyone has a common view and optimizing it based on using value stream mapping is a key tool in this process that helps to align the organization. The DP defines and documents the flow of code through the system, and value stream mapping the DP helps to identify bottlenecks and waste and other inefficiencies that can be addressed using DevOps techniques. Improving it will require a lot of organizational change management, but the DP will help everyone understand what processes are being changed at any one time and how they should start working differently.

The DP for a large organization with a tightly coupled architecture is a fairly complex concept to grasp. Therefore, in Chapter 2, we will start with the simplest example of a DP with one developer and will show the inefficiencies that can occur with one developer. Then, in Chapter 3, we will highlight the DevOps approaches that were designed to address those issues. We will also show the metrics you can start collecting to help you understand the magnitude of your inefficiencies so you can align your organization on fixing the issues that will provide the biggest benefit.

Once the basic construct of the DP is well understood, in Chapter 4 we will show how the complexity changes as you start scaling the DP from one developer to a team of developers. Having a team of developers working together on an application while keeping it close to release quality is a fundamental shift for most traditional organizations. It requires some different technical approaches by the developers, but it also requires a cultural shift that prioritizes keeping the code base stable over creating new features. This will be a big shift for most organizations, but it is very important because if you can't get the developers to respond to the feedback from the DP, then creating it will be of limited value.

The next big challenge large organizations have after they have had some success at the team level concerns how to scale DevOps across a large organization. They typically approach it by trying to get the rest of the organization to do what they did because of the benefits it provided. This overlooks the fact that the biggest barriers to

adoption are not technical, but instead involve organizational change management and getting people to work differently. The key to this adoption is helping the broader organization understand the principles, while providing as much flexibility as possible to allow them to develop and take ownership of their plans. In order to make this adoption of principles as flexible as possible, in Chapter 5 we will cover how to segment the work in large organizations into the smallest pieces possible to enable local control and ownership. For some organizations with loosely coupled architectures, this will result in a lot of small, independent teams where you only have to coordinate the work across tens of people. For other organizations with tightly coupled architectures that require large applications to be developed, qualified, and released together, this will require coordinating the work across hundreds of people. It is important to start by grouping applications into these types because the things you do to coordinate the work across tens of people will be different than the types of things you do to coordinate the work across hundreds of people. While small teams will always be more efficient and deploy more frequently, the process of documenting, automating, and continually improving DPs is much more important for coordinating work across hundreds of people because the inefficiencies across large organizations are much more pronounced.

In Chapter 6, we will provide a quick overview of the approaches that work well for large organizations with small teams that can work independently. This topic will not be covered in a lot of detail because most available DevOps material already covers this very well. In Chapter 7, we will start addressing the complexities of designing a DP for large, tightly-coupled systems. We will show how to break the problem into smaller more manageable pieces and then build those up into more complex releasable systems. In Chapter 8, we cover how to start optimizing these complex DPs, including metrics, to help focus changes in the areas where they will most help the flow through the system. In Chapter 9, we will review and highlight the differences between implementing improvements for small independent teams and for large complex systems.

Changing how a large organization works is going to take a while, and it is going to require changing how everyone both thinks about and does their actual work. A couple of things are important to consider when contemplating this type of organizational change: first, start where it provides the most benefit so you can build positive momentum, and second, find executives that are willing to lead the change and prioritize improvements that will optimize the DP instead of letting teams sub-optimize their segment of the DP.

Once the DP is in place, it provides a very good approach for transforming how you manage large and complex software projects. Instead of creating lots of management processes to track progress and align different teams, you use working code as the forcing function that aligns the organization. Requiring all the different Development teams to integrate their code on a regular basis and ensure it is working with automated testing forces them to align their software designs without a lot of management overhead.

The move to infrastructure as code, which was spearheaded by Jez Humble and David Farley and involves treating all aspects of the software development process with the same of rigor as application code, provided some major breakthroughs. It requires that the process for creating environments, deploying code, and managing databases be automated with code that is documented and tracked in a source code management (SCM) tool just like the application code. This move to infrastructure as code forces a common definition of environments and deployment processes across Development, QA, and Operations teams and ensures consistency on the path to production. Here again it is working code that helps to align these different groups.

Moving to infrastructure as code increases direct communication between Development and Operations, which is key to the success of all sorts of cultural and structural shifts DevOps requires. People no longer log on to computers and make changes that can't be tracked. Instead they work together on common scripts for making changes to the infrastructure that can be tracked in SCM tool. This requires

them, at minimum, to document any changes they are making so everyone can see what they are doing, and ideally it forces them to communicate directly about the changes they are making so they can ensure those changes will work in every stage in the DP all the way out to production. Having to use common code and common tools forces the collaboration. The effect that this collaboration has on efficiency cannot be underestimated. Since the teams are aligned by having to ensure their code works together on a daily basis, management processes do not need to be put in place to address those issues. Software is notoriously hard to track well with management processes. Getting status updates everywhere doesn't work that well and takes a lot of overhead. It is more efficient if the teams resolve issues in real time. Additionally, it is much easier to track progress using the DP because instead of creating lots of different managerial updates, everyone can track the progress of working code as it moves down the pipeline.

This approach of a rigorous DP with infrastructure as code and automated testing gating code progression is significantly different from the approach ITIL uses for configuration management. Where the ITIL processes were designed to ensure predictability and stability, the DevOps changes have been driven by the need to improve speed while maintaining stability. The biggest changes are around configuration management and approval processes. The ITIL approach has very strict manual processes for any changes that occur in the configuration of production. These changes are typically manually documented and approved in a change management tool with tickets. The approved changes are then manually implemented in production. This approach helped improve stability and consistency, but slowed down flow by requiring lots of handoffs and manual processes. The DevOps approach of infrastructure as code with automated testing as gates in the DP enables better control of configuration and more rigors in the approval process, while also dramatically improving speed. It does this by automating the process with code and having everything in the SCM tool. The code change being proposed is documented by the script change in the SCM. The approval criteria

for accepting the change is documented by automated tests that are also in the SCM. Additionally, you know exactly what change was implemented because it was done with the automation code under revision control. The whole approach puts everything required for change management in one tool with automation that is much easier and quicker to track. It also improves the rigors in the approval processes by requiring the people who traditionally approve the changes to document their criteria via automated tests instead of just using some arbitrary management decision for each change.

This approach provides some huge benefits for auditing and regulatory compliance. Where before the audit team would have to track the manual code changes, approval processes, and implementations in different tools, it is now all automated and easily tracked in one place. It dramatically improves compliance because computers are much better than humans at ensuring the process is followed every time. It is also easier for the auditing team because all the changes are documented in a (SCM) tool that is designed for automatically tracking and documenting changes.

These changes are dramatically improving the effectiveness of large organizations because they improve the flow of value while maintaining stability. Most importantly, though, is that setting up and optimizing a DP requires removing waste and inefficiencies that have existed in your organization for years. In order to improve the flow, you will end up addressing lots of inefficiencies that occur in coordinating work across people. The productivity of individuals will be improved by better quality and faster feedback while they are writing code, but the biggest benefits will come from addressing the issues coordinating the work within teams, across teams, and across organizations. It will require technical implementations and improvement, but by far the biggest challenge is getting people to embrace the approaches and change how they work on a day-to-day basis. These changes will be significant, but the benefits will be dramatic.

Summary

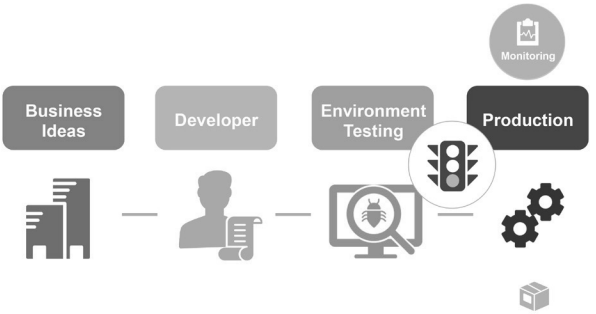
As software becomes the basis of competition, how we currently manage software limits the kinds of quick responses that businesses require. This is where DevOps steps in. It is all about improving speed while maintaining all aspects of quality. As businesses embark on DevOps journeys, though, they are finding that there are myriad ideas out there about what DevOps is and how it is defined. As this book will address, most large organizations don't have a good framework for putting all these different ideas into context as they start their DevOps journey. This makes it difficult to get everyone working together on changes that will improve the end-to-end system. People working in a large organization need to be aligned on what they are going to build and need to find ways to prioritize improvement or else they won't implement DevOps in ways that will deliver the expected results. As this book will show, documenting, automating, and optimizing DPs in large software/IT organizations improves efficiency and effectiveness and offers a very good approach for transforming how you manage large and complex software projects.

Chapter 2

THE BASIC DEPLOYMENT PIPELINE

The DP in a large organization can be a complex system to understand and improve. Therefore, it makes sense to start with a very basic view of the DP, to break the problem down into its simplest construct and then show how it scales and becomes more complex when you use it across big, complex organizations. The most basic construct of the DP is the flow of a business idea to development by one developer through a test environment into production. This defines how value flows through software/IT organizations, which is the first step to understanding bottlenecks and waste in the system. Some people might be tempted to start the DP at the developer, but I tend to take it back to the flow from the business idea because we should not overlook the amount of requirements inventory and inefficiencies that waterfall planning and the annual budgeting process drive into most organizations.

The first step in the pipeline is communicating the business idea to the developer so they can create the new feature. Then, once the new feature is ready, the developer will need to test it to ensure that it is working as expected, that the new code has not broken any existing functionality, and that it has not introduced any security holes or impacted performance. This requires an environment that is representative of production. The code then needs to be deployed into the test environment and tested. Once the testing ensures the new code is working as expected and has not broken any other existing functionality, it can be deployed into production, tested, and released. The final step is monitoring the application in production to ensure it is working as expected. In this chapter, we will review each step in this process, highlighting the inefficiencies that frequently occur. Then, in Chapter 3, we will review the DevOps practices that were developed to help address those inefficiencies.



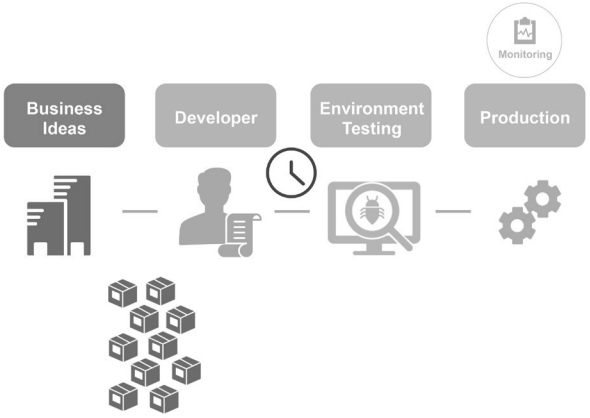
Requirements

The first step in the DP is progressing from a business idea to work for the developer to create the new feature. This usually involves creating a requirement and planning the development to some extent. The first problem large organizations have with flow of value through their DP is that they tend to use waterfall planning. They do this because they use waterfall planning for every other part of their business so they just apply the same processes to software. Software, however, is unlike anything else most organizations manage in three ways. First, it is much harder to plan accurately because everything you are asking your teams to do represents something they are being asked to do it for the first time. Second, if software is developed correctly with a rigorous DP, it is relatively quick and inexpensive to change. Third, as an industry we are so poor at predicting our customers' usage that over 50% of all software developed is never used or does not meet its business intent. Because of these unique characteristics of software, if you use waterfall planning, you end up locking in your most flexible and valuable asset in order to deliver features that won't ever be used or won't deliver the intended business results. You also use up a significant amount of your capacity planning instead of delivering real value to your business.

Organizations that use waterfall planning also tend to build up lots of requirements inventory in front of the developer. This inventory tends to slow down the flow of value and creates waste and inefficiencies in the process. As the Lean manufacturing efforts have

clearly demonstrated, wherever you have excess inventory in the system tends to drive waste in terms of rework and expediting. If the organization has invested in creating the requirements well ahead of when they are needed, when the developer is ready to engage, the requirement frequently needs to be updated to answer any questions the developer might have and/or updated to respond to changes in the market. This creates waste and rework in the system.

The other challenge with having excess inventory of requirements in front of the developer is that as the marketplace evolves, the priorities should also evolve. This leads to the organization having to reprioritize the requirements on a regular basis or, in the worst case, sticking to a committed plan and delivering features that are less likely to meet the needs of the current market. If these organizations let the planning process lock them into committed plans, it creates waste by delivering lower value features. If the organizations reprioritize a large inventory of requirements, they will likely deprioritize requirements that the organization has invested a lot of time and energy in creating. Either way, excess requirements inventory leads to waste.



Test Environment

The next step is getting an environment where the new feature can be deployed and tested. The job of providing environments typically belongs to Operations, so they frequently lead this effort. In small

organizations using the cloud, this can be very straightforward and easy. In large organizations using internal datacenters, this can be a very complex and timely process that requires working through extensive procurement and approval processes with lengthy handoffs between different parts of the organization. Getting an environment can start with long procurement cycles and major operational projects just to coordinate the work across the different server, storage, networking, and firewall teams in Operations. This is frequently one of the biggest pain points that cause organizations to start exploring DevOps.

There is one large organization that started their DevOps initiative by trying to understand how long it would take to get up Hello World! in an environment using their standard processes. They did this to understand where the biggest constraints were in their organization. They quit this experiment after 250 days even though they still did not have Hello World! up and running because they felt they had identified the biggest constraints. Next, they ran the same experiment in Amazon Web Services and showed it could be done in two hours. This experiment provided a good understanding of the issues in their organization and also provided a view of what was possible.

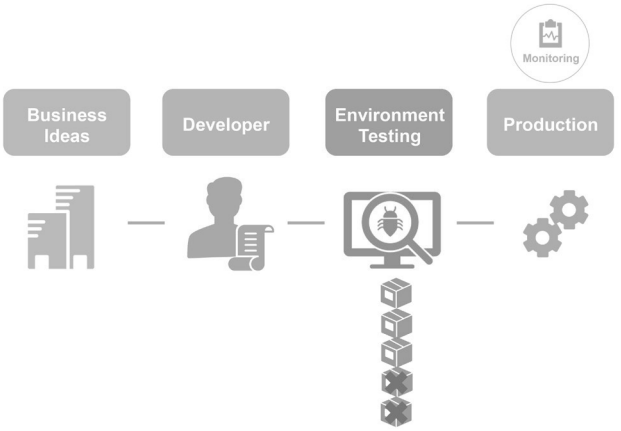
Testing and Defect Fixing

Once the environment is ready, the next step is deploying the code with the new feature into the test environment and ensuring it works as expected and does not break any existing functionality. This step should also ensure that there were no security or performance issues created by the new code. Three issues typically plague traditional organizations at this stage in their DP: repeatability of test results, the time it takes to run the tests, and the time it takes to fix all the issues.

Repeatability of the results is a big source of inefficiency for most organizations. They waste time and energy debugging and trying to find code issues that end up being problems with the environment, the code deployment, or even the testing process. This makes it

extremely difficult to determine when the code is ready to flow into production and requires a lot of extra triaging effort for the organization. Large, complex, tightly coupled organizations frequently spend more time setting up and debugging these environments than they do writing code for the new capabilities.

This testing is typically done with expensive and time-consuming manual tests that are not very repeatable. This is why it's essential to automate your testing. The time it takes to run through a full cycle of manual testing delays the feedback to developers, which results in slow rework cycles, which reduces flow in the DP. The time and expense of these manual test cycles also forces organizations to batch lots of new features together into major releases, which slows the flow of value and makes the triage process more difficult and inefficient.



The next challenge in this step is the time and effort it takes to remove all the defects from the code in the test environment and to get the applications up to production level quality. In the beginning, the biggest constraint is typically the time it takes to run all the tests. When this takes weeks, the developers can typically keep up with fixing defects at the rate at which the testers are finding them. This changes once the organization moves to automation where all the testing can be run in hours, at which point the bottleneck tends to move toward the developers ability to fix all the defects and get the code to production levels of quality.

Once an organization gets good at providing environments or is just adding features to an application that already has environments set up, reaching production level quality is frequently one of the biggest challenges to releasing code on a more frequent basis. I have worked with organizations that have the release team leading large cross-organizational meetings to get applications tested, fixed, and ready for production. They meet every day to review the testing progress to see when it will be done so they are ready to release to production. They track all the defects and fixes so they can make sure the current builds have production level quality. Frequently, you see these teams working late on a Friday night to get the build ready for offshore testing over the weekend only to find out Saturday morning that all the offshore teams were testing with the wrong code or a bad deployment, or the environment was misconfigured in some way. This process can drive a large amount of work into the system and is so painful that many organizations choose to batch very large, less frequent releases to limit the pain.

Production Deployment

Once all the code is ready, the next step is to deploy the code into production for testing and release to the customer. Production deployment is an Operations led effort, which is important because Operations doesn't always take the lead in DevOps transformations, but when you use the construct of the DP to illustrate how things work, it becomes clear that Operations is essential to the transformation and should lead certain steps to increase efficiency in the process. It is during this step that organizations frequently see issues with the application for the first time during the release. It is often not clear if these issues are due to code, deployment, environments, testing, or something else altogether. Therefore, the deployment of large complex systems frequently requires large cross-organizational launch calls to support releases. Additionally, these deployment processes themselves can require lots of time and resources for manual implementations. The amount of time, effort, and angst associated

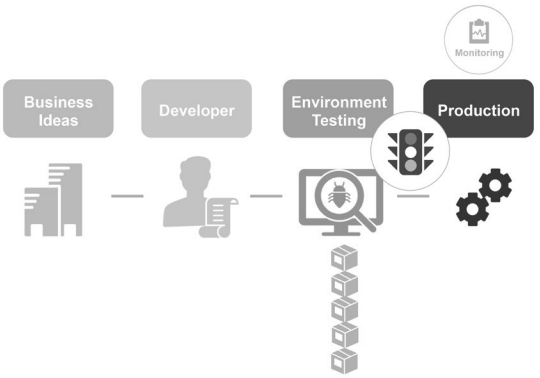
with this process frequently pushes organizations into batching large amounts of change into less frequent releases.

Monitoring and Operations

Monitoring is typically another Operations-led effort since they own the tools that are used to monitor production. Frequently, the first place in the DP that monitoring is used is in production. This is problematic because when code is released to customers, developers haven't been able to see potential problems clearly before the customer experience highlights it. If Operations works with Development to move monitoring up the pipeline, potential problems are caught earlier and before they impact the customer.

When code is finally released to the customers and monitored to ensure it is working as expected, then ideally there shouldn't be any new issues caught with monitoring in production if all the performance and security testing was complete with good coverage. This is frequently not the case in reality. For example, I was part of one large release into production where we had done extensive testing going through a rigorous release process, only to have it immediately start crashing in production as a result of an issue we had never seen before. Every time we pointed customer traffic to the new code base, it would start running out of memory and crashing. After several tries and collecting some data, we had to spend several hours rolling back to the old version of the applications. We knew where the defect existed, but even as we tried debugging the issues, we couldn't reproduce it in our test environments. After a while, we decided we couldn't learn any more until we deployed into production and used monitoring to help locate the issue. We deployed again, and the monitoring showed us that we were running out of memory and crashing. This time the developers knew enough to collect more clues to help them identify the issue. It turns out a developer was fixing a bug that was not wrapping around a long line of text correctly. The command the developer had used worked fine in all our testing, but in production we realized that IE8 localized to Spanish had a defect that would turn this command into a floating point instead of

an integer, causing a stack overflow. This was such a unique corner case, we would not have considered testing for it. Additionally, even if we had considered it, running all our testing on different browsers with different localizations would have become cost prohibitive. It is issues like this that remind us that the DP is not complete until the new code has been monitored in production and is behaving as expected.



Summary

Understanding and improving a complex DP in a large organization can be a complicated process. Therefore, it makes sense to start by exploring a very simple DP with one developer and understanding the associated challenges. This process starts with the business idea being communicated to the developer and ends with working code in production that meets the needs of the customer. There are lots of things that can and do go wrong in large organizations, and the DP provides a good framework for putting those issues in context. In this chapter, we introduced the concept and highlighted some typical problems. Chapter 3 will introduce the DevOps practices that are designed to address issues at each stage in the pipeline and provide some metrics that you can use to target improvements that will provide the biggest benefits.

Chapter 3

OPTIMIZING THE BASIC DEPLOYMENT PIPELINE

Setting up your DP and using DevOps practices for increasing its throughput while maintaining or improving quality is a journey that takes time for most large organizations. This approach, though, will provide a systematic method for addressing inefficiencies in your software development processes and improving those processes over time. We will look at the different types of work, different types of waste, and different metrics for highlighting inefficiencies. We will start there because it is important to put the different DevOps concepts, metrics, and practices into perspective so you can start your improvements where they will provide the biggest benefits and start driving positive momentum for your transformation.

The technical and cultural shifts associated with this will change how everyone works on a day-to-day basis. The goal is to get people to accept these cultural changes and embrace different ways of working. For example: As an Operations person, I have always logged into a server to debug and fix issues on the fly. Now I can log on to debug, but the fix is going to require updating and running the script. This is going to be slower at first and will feel unnatural to me, but the change means I know, as does everyone else, that the exact state of the server with all changes are under version control, and I can create new servers at will that are exactly the same. Short-term pain for long-term gain is going to be hard to get some people to embrace, but this is the type of cultural change that is required to truly transform your development processes.

Additionally, there are lots of breakthroughs coming from the field of DevOps that will help you address issues that have been plaguing your organization for years that were not very visible while operating at a low cadence. When you do one deployment a month, you don't see the issues repeating enough to see a common cause that needs to

be fixed. When you do a deployment each day, you see a pattern that reveals the things that need fixing. When you are deploying manually on a monthly basis, you can use brute force, which takes up a lot of time, requires a lot of energy, and creates a lot of frustration. When you deploy daily, you can no longer use brute force. You need to automate to improve frequency, and that automation allows you to fix repetitive issues.

As you look to address inefficiencies, it is important to understand that there are three different kinds of work with software that require different approaches to eliminate waste and improve efficiency. First, there is new and unique work, such as the new features, new applications, and new products that are the objective of the organization. Second, there is triage work that must be done to find the source of the issues that need to be fixed. Third, there is repetitive work, which includes creating an environment, building, deploying, configuring databases, configuring firewalls, and testing.

Since the new and unique work isn't a repetitive task, it can't be optimized the way you would a manufacturing process. In manufacturing, the product being built is constant so you can make process changes and measure the output to see if there was an improvement. With the new and unique part of software you can't do that because you are changing both the product and the process at the same time. Therefore, you don't know if the improvement was due to the process change or just a different outcome based on processing a different type or size of requirement. Instead the focus here should be on increasing the feedback so that people working on these new capabilities don't waste time and energy on things that won't work with changes other people are making, won't work in production, or don't meet the needs of the customer. Providing fast, high-quality feedback helps to minimize this waste. It starts with feedback in a production-like environment with their latest code working with everyone else's latest code to ensure real-time resolution of those issues. Then, ideally, the feedback comes from the customer with code in production as soon as possible. Validating with the customer is done to address the fact that 50% of new software features are

never used or do not meet their business intent. Removing this waste requires getting new features to the customers as fast as possible to enable finding which parts of the 50% are not meeting their business objective so the organization can quit wasting time on those efforts.

In large software organizations, triaging and localizing the source of the issue can consume a large amount of effort. Minimizing waste in this area requires minimizing the amount of triage required and then designing processes and approaches that localize the source of issues as quickly as possible when triage is required. DevOps approaches work to minimize the amount of triage required by automating repetitive tasks for consistency. DevOps approaches are also designed to improve the efficiency of the triage process by moving to smaller batch sizes, resulting in fewer changes needing to be investigated as potential sources of the issue.

The waste with repetitive work is different. DevOps moves to automate these repetitive tasks for three reasons. First, it addresses the obvious waste of doing something manually when it could be automated. Automation also enables the tasks to be run more frequently, which helps with batch sizes and thus the triage process. Second, it dramatically reduces the time associated with these manual tasks so that the feedback cycles are much shorter, which helps to reduce the waste for new and unique work. Third, because the automated tasks are executed the same way every time, it reduces the amount of triage required to find manual mistakes or inconsistencies across environments.

DevOps practices are designed to help address these sources of waste, but with so many different places that need to be improved in large organizations, it is important to understand where to start. The first step is documenting the current DP and starting to collect data to help target the bottlenecks in flow and the biggest sources of waste. In this chapter we will walk through each step of the basic DP and will review which metrics to collect to help you understand the magnitude of issues you have at each stage. Then, we will describe the DevOps approaches people have found effective for addressing

the waste at that stage. Finally, we will highlight the cultural changes that are required to get people to accept working differently.

This approach should help illustrate why so many different people have different definitions of DevOps. It really depends what part of the elephant they are seeing. For any given organization, the constraint in flow may be the planning/requirements process, the development process, obtaining consistent environments, the testing process, or deploying code. Your view of the constraint also potentially depends on your role in the organization. While everything you are hearing about DevOps is typically valid, you can't simply copy the rituals because it might not make sense for your organization. One organization's bottleneck is not another organization's bottleneck so you must focus on applying the principles!

Requirement/Planning

Here we are talking about new and unique work, not repetitive work, so fixing it requires fast feedback and a focus on end-to-end cycle time for ultimate customer feedback.

For organizations trying to better understand the waste in the planning and requirements part of their DP, it is important to understand the data showing the inefficiencies. It may not be possible to collect all the data at first, but don't let this stop you from starting your improvements. As with all of the metrics we describe, get as much data as you can to target issues and start your continuous improvement process. It is more important to start improving than it is to get a perfect view of your current issues. Ideally, though, you would want to know the answers to the following questions:

- *What percentage of the organizations capacity is spent on documenting requirements and planning?*
- *What is the amount of requirements inventory waiting for development, roughly, in terms of days of supply?*

- *What percentages of the requirements are reworked after originally defined?*
- *What percentages of the delivered features are being used by the customers and are achieving the expected business results?*

Optimizing this part of the DP requires moving to a just-in-time approach to documenting and decomposing requirements only to the level required to support the required business decisions while limiting the commitment of long-term deliveries to a subset of the overall capacity. The focus here is to limit the inventory of requirements as much as possible. Ideally this would wait until the developer is ready to start working on the requirement before investing in defining the feature. This approach minimizes waste because effort is not exerted until you know for sure it is going to be developed. It also enables quick responsiveness to changes in the market because great new ideas don't have to wait in line behind all the features that were previously defined.

While this is the ideal situation, it is not always possible because organizations frequently need a longer-range view of when things might happen in order to support different business decisions. For example, you might ask yourself, "Do I need to ramp up hiring to meet schedule, or should I build the manufacturing line because a product is going to be ready for a launch?" The problem is that most organizations create way more requirements inventory a long way into the future than is needed to support their business decisions. They want to know exactly what features will be ready when using waterfall planning because that is what they do for every other part of the business. The problem is that this approach drives a lot of waste into the system and locks in to a committed plan what should be your most flexible asset. Additionally, most organizations push their software teams to commit to 100% of their capacity, meaning they are not able to respond to changes in the marketplace or discoveries during development. This is a significant source of waste in a lot of organizations.

I have worked with one organization that moved to a more just-in-time approach for requirements and that has transformed their planning processes from taking 20% or more of their capacity to less than 5%. They eliminated waste and freed up 15% of the capacity of their organization to focus on creating value for the business. This was done by limiting long-term commitments of over a year to less than 50% of capacity and committing additional capacity in shorter timeframe horizons. The details of how this worked are in Chapter 5 of Leading the Transformation by Gary Gruver and Tommy Mouser. This was a big shift that freed up more capacity, and it also improved the speed of value through the system because new ideas could move quickly into development if they were of the highest priority instead of waiting in queue behind a lot of lower-priority ideas that were previously planned.

This move is a big cultural change for most organizations. It requires software/IT and business executives to think differently about how they manage software. They really need to change their focus from optimizing the system for accuracy in plans to optimizing it for throughput of value for the customer. They need to be clear about the business decisions they need to support and work with the organization to limit the investment in requirements just to the level of detail required to support those decisions.

Environments

For many organizations, like the one described in Chapter 2, the time it takes for Operations to create an environment for testing is one of the lengthiest steps in the DP. Additionally, the consistency between this testing environment and production is so lacking that it requires finding and fixing a whole new set of issues at each stage of testing in the DP. Creating these environments is one of the main repetitive tasks that can be documented, automated, and put under revision control. The objective here is to be able to quickly create environments that provide consistent results across the DP. This is done through a movement to infrastructure as code, which has the additional advantage of documenting everything about the

environments so it is easier for different parts of the organization to track and collaborate on changes.

To better understand the impact environment issues are having on your DP, it would be helpful to have the following data:

- *time from environment request to delivery*
- *how frequently new environments are required*
- *the percent of time environments need fixing before acceptance*
- *the percent of defects associated with code vs. environment vs. deployment vs. database vs. other at each stage in the DP*

One of the biggest improvements coming out of the DevOps movement concerns the speed and consistency of environments, deployments, and databases. This started with Continuous Delivery by Jez Humble and David Farley. They showed the value of infrastructure as code, where all parts of the environment are treated with the same rigor and controls as the application code. The process of automating the infrastructure and putting it under version control has some key advantages. First, the automation ensures consistency across different stages and different servers in the DP. Second, the automation supports the increased frequency that is required to drive to smaller batch sizes and more frequent deployments. Third, it provides working code that is a well-documented definition of the environments that everyone can collaborate on when changes are required to support new features.

Technical solutions in this space are quickly evolving because organizations are seeing that getting control of their environments provides many benefits. Smart engineers around the world are constantly inventing new ways to make this process easier and faster. Cloud capabilities, whether internal or external, tend to help a lot with speed and consistency. New scripting capabilities from Chef, Puppet, Ansible, and others help with getting all the changes in scripts under source control management. There have also been

breakthroughs with containers that are helping with speed and consistency. The "how" in this space is evolving quickly because of the benefits the solutions are providing, but the "what" is a lot more consistent. For environments, you don't want the speed of provisioning to be a bottleneck in your DP. You need to be able to ensure consistency of the environment, deployment process, and data across different stages of your DP. You need to be able to qualify infrastructure code changes efficiently so your infrastructure can move as quickly as your applications. Additionally, you need to be able to quickly and efficiently track everything that changes from one build and environment to the next.

Having Development and Operations collaborate on these scripts for the entire DP is essential. The environments across different stages of the DP are frequently different sizes and shapes, so often no one person understands how a configuration change in the development stage should be implemented in every stage through production. If you are going to change the infrastructure code, it has to work for every stage. If you don't know how it should work in those stages, it forces necessary discussions. If you are changing it and breaking other stages without telling anyone, the SCM will find you out and the people managing the DP will provide appropriate feedback. Working together on this code is what forces the alignment between Development and Operations. Before this change, Development would tend to make a change to fix their environment so their code would work, but they wouldn't bother to tell anyone or let people know that in order for their new feature to work, something would have to change in production. It was release engineering's job to try and figure out everything that had changed and how to get it working in production. With the shift to infrastructure as code, it is everyone's responsibility to work together and clearly document in working automation code all of the changes.

This shift to infrastructure as code also has a big impact on the ITIL and auditing processes. Instead of the ITIL processes of documenting configuration of a change manually in a ticket, it is all documented in code that is under revision control in a SCM tool.

The SCM is designed to make it easy to track any and all changes automatically. You can look at any server and see exactly what was changed by who and when. Combine this with automated testing that can tell you when the system started failing, and you can quickly get to the change that caused the problem. This localization gets easier when the cycle time between tests limits this to a few changes to look through.

Right now, the triage process takes a long time to sort through clues to find the change that caused the problem. It is hard to tell if it is a code, environment, deploy, data, or test problem. and currently the only thing under control for most organizations is code. Infrastructure as code changes that and puts everything under version control that is tracked. This eliminates server-to-server variability and enables version control of everything else. This means that the process for making the change and documenting the change are the same thing so you don't have to look at the documentation of the change in one tool to see what was approved and then validate that it was really done in the other tool. You also don't have to look at everything that was done in one tool and then go to the other tool to ensure it was documented. This is what they do during auditing. The other thing done during auditing is tracking to ensure everyone is following the manual processes every time–something that humans do very poorly, but computers do very well. When all this is automated, it meets the ITIL test of tracking all changes, and it makes auditing very easy. The problem is that the way DevOps is currently described to process and auditing teams makes them dig in their heels and block changes when instead they should be championing those changes. To avoid this resistance to these cultural changes, it is important to help the auditing team understand the benefits it will provide and include them in defining how the process will work. This will make it easier for them to audit, and they will know where to look for the data they require.

Using infrastructure as code across the DP also has the benefit of forcing cultural alignment between Development and Operations. When Development and Operations are using different tools and

processes for creating environments, deploying code into those environments, and managing databases, they tend to find lots of issues releasing new code into production. This can lead to a great deal of animosity between Development and Operations. As they start using the same tools, and more specifically the same code, you will likely find that making the code work in all the different stages of the DP forces them to collaborate much more closely. They need to understand each other's needs and the differences between the different stages much better. They also need to agree that any changes to the production environments start at the beginning of the DP and propagate through the system just like the application code. Over time, you will likely find that this working code is the forcing function that starts the cultural alignment between Development, Operations, and all the organizations in between. This is a big change for most large organizations. It requires that people quit logging in to servers and making manual changes. It requires an investment in creating automation for the infrastructure. It also requires everyone to use common tools, communicate about any infrastructure changes that are required, and document the changes with automated scripts. It requires much better communication across the different silos than exists in most organizations.

Organizations doing embedded development typically have a unique challenge with environments because the firmware/software systems are being developed in parallel with the actual product so there is very little, if any, product available for early testing. Additionally, even when the product is available, it is frequently difficult to fully automate the testing in the final product. These organizations need to invest in simulators to enable them to test the software portions of their code as frequently and cheaply as possible. They need to find or create a clean architectural interface between the software parts of their code and the low-level embedded firmware parts. Code is then written that can simulate this interface running on a blade server so they can test the software code without the final product. The same principle holds true for the low-level embedded firmware, but this testing frequently requires validating the interactions of this code

with the custom hardware in the product. For this testing, they need to create emulators that support testing of the hardware and firmware together without the rest of the product.

This investment in simulators and emulators is a big cultural shift for most embedded organizations. They typically have never invested to create these capabilities and instead just do big bang integrations late in the product lifecycle that don't go well. Additionally, those that have created simulators or emulators have not invested in continually improving these capabilities to ensure they can catch more and more of the defects over time. These organizations need to make the cultural shift to more frequent test cycles just like any other DevOps organization, but they can't do that if they don't have test environments they can trust for finding code issues. If the organization is not committed to maintaining and improving these environments, the organization tends to loose trust and quit using them. When this happens, they end up missing a key tool for transforming how they do embedded software and firmware development.

Testing

The testing, debug, and defect fixing stage of the DP is a big source of inefficiencies for lots of organizations. To understand the magnitude of the problem for your DP, it would be helpful to have the following data:

- *the time it takes to run the full set of testing*
- *the repeatability of the testing (false failures)*
- *the percent of defects found with unit tests, automated system tests, and manual tests*
- *the time it takes the release branch to meet production quality*
- *approval times*
- *batch sizes or release frequency at each stage*

The time it takes for testing is frequently one of the biggest bottlenecks in flow in organizations starting on the DevOps journey. They depend on slow-running manual tests to find defects and support their release decisions. Removing or reducing this bottleneck is going to require moving to automated testing. This automated testing should include all aspects of testing required to release code into production: regression, new functionality, security, and performance. Operations should also work to add monitoring or other operational concerns to these testing environments to ensure issues are found and feedback is given to developers while they are writing code so they can learn and improve. Automating all the testing to run within hours instead of days and weeks is going to be a big change for most organizations. The tests need to be reliable and provide consistent results if they are going to be used for gating code. You should run them over and over again in random order against the same code to make sure they provide the same result each time and can be run in parallel on separate servers. Make sure the test automation framework is designed so the tests are maintainable and triageable. You are going to be running and maintaining thousands of automated tests running daily, and if you don't think through how this is going to work at scale, you will end up dying under the weight of the test automation instead of reaping its benefits. This requires a well-designed automation framework that is going to require close collaboration between Development and QA.

It is important to make sure the tests are designed to make the triage process more efficient. It isn't efficient from a triage perspective if the system tests are finding lots of environment or deployment issues. If this happens, you should start designing specific post-deployment tests to find and localize these issues quickly. Then once the post-deployment tests are in place, make sure they are passing and the environments are correct before starting any system testing. This approach improves the triage efficiency by separating code and infrastructure issues with the design of the testing process.

Automated testing and responding to feedback is going to be a big cultural shift for most organizations. The testing process is going to

have to move from manually knowing how to test the applications to using leading edge programming skills to automate testing of the application. These are skills that don't always exist in organizations that have traditionally done manual testing. Therefore, Development and the test organization are going to have to collaborate to design the test framework. Development is going to have to modify how they write code so that automated testing will be stable and maintainable. And probably the biggest change is to have the developers respond to test failures and keep build stability as their top priority.

If you can't get this shift to happen, it probably doesn't make sense to invest in building out complex DPs that won't be used. The purpose of the automated testing is not to reduce the cost of testing, but to enable the tests to be run on a more frequent basis to provide feedback to developers in order to reduce waste in new and unique work. If they are not responding to this feedback, then it is not helping. Therefore, it is important to start this cultural shift as soon as possible. Don't write a bunch of automated tests before you start using them to gate code. Instead, write a few automated build acceptance tests (BATs) that define a very minimal level of stability. Make sure everyone understands that keeping those tests passing on every build is job one. Watch this process very carefully. If it is primarily finding test issues, review and redesign your test framework. If it is primarily finding infrastructure issues, start designing post-deployment tests to ensure stability before running any system test looking for code issues. If it is primarily finding code issues, then you are on the right track and ready to start the cultural transformation of having the developers respond to feedback from the DP. The process of moving to automated tests gating code is going to be a big cultural shift, but it is probably one of the most important steps in changing how software is developed.

Testing more frequently on smaller batches of changes makes triage and debugging much easier and more efficient. The developers receive feedback while they are writing the code and engaged in that part of the design instead of weeks later when they have moved on to something else. This makes it much easier for them to learn

from their mistakes and improve instead of just getting beat up for something they don't even remember doing. Additionally, there are fewer changes in the code base between the failure and the last time it passed, so you can quickly localize the potential sources of the problem.

The focus for automated testing really needs to be on increasing the frequency of testing and ensuring the organization is quickly responding to failures. This should be the first step for two reasons. First, it starts getting developers to ensure the code they are writing is not breaking existing functionality. Second, and most importantly, it ensures that your test framework is maintainable and triagable before you waste time writing tests that won't work over the long term.

I worked with one organization that was very proud of the fact that they had written over one thousand automated tests that they were running at the end of each release cycle. I pointed out that this was good, but to see the most value, they should start using them in the DP every day, gating builds where the developers were required to keep the builds green. They should also make sure they started with the best, most stable tests because if the red builds were frequently due to test issues instead of code issues, then the developers would get upset and disengage from the process. They spent several weeks trying to find reliable tests out of the huge amount available. In the end, they found out that they had to throw out all the existing tests because they were not stable, maintainable, or triagable. Don't make this same mistake! Start using your test automation as soon as possible. Have the first few tests gating code on your DP, and once you know you have a stable test framework, start adding more tests over time.

Once you have good test automation in place that is running in hours instead of days or weeks, the next step to enabling more frequent releases is getting and keeping trunk much closer to production-level quality. If you let lots of defects build up on trunk while you are waiting for the next batch release, then the bottleneck in your DP

will be the amount of time and energy it takes to fix all the defects before releasing into production. The reality is that to do continuous deployment, trunk has to be kept at production levels of quality all the time. This is a long way off for most organizations, but the benefit of keeping trunk closer to production-level quality is worth the effort. It enables more frequent, smaller releases because there is not as big an effort to stabilize a release branch before going into production. It also helps with the localization of issues because it is easier to identify changes in quality when new code is integrated. Lastly, while you may still have some manual testing in place, it ensures that your testers are as productive as possible while working on a stable build. This might not be your bottleneck if you start with a lot of manual testing because the developers can fix defects as quickly as the testers can find them. However, this starts to change as you add more automated tests. Watch for this shift, and be ready to move your focus as the bottleneck changes over time.

This transition to a more stable trunk is a journey that is going to take some time. Start with a small set of tests that will define the minimal level of stability that you will ever allow in your organization. These are your BATs. If these fail due to a change, then job one is fixing those test failures as quickly as possible. Even better, you should automatically block that change from reaching trunk. Then over time, you should work to improve the minimal level of stability allowed on trunk by farming your BAT tests. Have your QA organization help identify issues they frequently find in builds that impact their ability to manually test effectively. Create an automated test to catch this in real time. Add it to the BAT set, and never do any manual testing on a build until the all the automated tests are passing. Look for major defects that are getting past the current BAT tests, and add a test to fill the hole. Look for long running BAT tests that are not finding defects, and remove them so you have time to add more valuable tests. This is a constant process of farming the BAT test that moves trunk closer to release quality over time.

If you are going to release more frequently with smaller batches, this shift to keeping trunk stable and closer to release quality is required.

It is also going to be a big shift for most organizations. Developers will need to bring code in without breaking existing functionality or exposing their code to customers until it is done and ready for release. Typically, organizations release by creating a release branch where they finalize and stabilize the code. Every project that is going to be in a release needs to have their code on trunk when the release branches. This code is typically brought in with the new features exposed to the customer ready for final integration testing. For lots of organizations, the day they release branch is the most unstable day for trunk because developers are bringing in last minute features that are not ready and have not been tested with the rest of the latest code. This is especially true for projects the business wants really badly. These projects tend to come in with the worst quality, which means every other project on the release has to wait until the really bad project is ready before the release branch can go to production. This type of behavior tends to lead to longer release branches and less frequent releases. To address this, the organization needs to start changing their definition of done. The code can and should be brought in but not exposed to the customer until it meets the new definition of done. If the organization is going to move to releasing more frequently, the new definition of done needs to change to include the following: all the stories are signed off, the automated testing is in place and passing, and there are no known open defects. This will be a big cultural shift that will take some time.

The final step in this stage of the DP is the approval for moving into production. For some organizations that are tightly regulated, this requires getting manual approval by someone in the management chain, which can take up to days to get. For organizations that are well down the path to continuous deployment, this can be the biggest bottleneck in the flow of code. To remove this bottleneck, highly regulated organizations move to have the manager who was doing the manual approval work with testers document their approval criteria with automated tests. For less regulated environments, having the developer take ownership and responsibility for quickly

resolving any issues found in productions can eliminate the management approval process.

There are lots of changes that can help improve the flow at this stage of the DP. The key is to make sure you are prioritizing improvements that will do the most to improve the flow. So, start with the bottleneck and fix it, then identify and fix the next bottleneck. This is the key to improving flow. If your test cycle is taking six weeks to run and your management approval takes a day, it does not make any sense to take on the political battle of convincing your organization that DevOps means it needs to let developers push code into production. If, on the other hand, testing takes hours, your trunk is always at production levels of quality, and your management approval takes days, then it makes sense to address the approval barriers that are slowing down the flow of code. It is important to understand the capabilities of your organization and the current bottlenecks before prioritizing the improvements.

Production Release

The next step in the basic DP is the release into production. Ideally, you would have found and fixed all the issues in the test stage so that this is a fairly automated and simple process. Realistically, this is not the case for most organizations. To better understand the source and magnitude of the issues at this stage, it is helpful to look at the following metrics:

- *the time and effort required to deploy and release into production*
- *the number of issues found during release and their source (code, environment, deployment, test, data, etc…)*

If you are going to release code into production with smaller more frequent releases, you can't have a long drawn out release process requiring lots of resources. Many organizations start with teams of Operations people deploying into a datacenter with run books and manual processes. This takes a lot effort and is often plagued with

manual errors and inconsistencies. DevOps addresses this by automating the release process as the final step in the DP. The process has been exercised and perfected during earlier stages in the DP and production is just the last repeat of the process. This automation ensures consistency and greatly reduces the amount of time and people required for release.

The next big challenge a lot of organizations have during the release process is that they are finding issues during the release process that they did not discover earlier in the DP. It is important to understand the source of these issues so the team can start addressing the reasons they were not caught before release into production. As much as possible, you should be using the same tools, processes, and scripts in the test environment as in the production environment. The test environment is frequently a smaller version of production, so it is not exact, but as much as possible you should work to abstract those differences out of the common code that that defines the environment, deploys the code, and configures the database. If you are finding a lot of issues associated with these pieces, start automating these processes and architect for as much common code across the DP as possible. Also, once you have this automation in place, any patches for production should start at the front end of the pipeline and flow through the process just like the application code.

Organizations with large complex deployments also frequently struggle with the triage process during the launch call. A test will fail, but it is hard to tell if it is due to an environment, deployment, database, code, or test issue. The automated testing in the deployment process should be designed to help in this triage process. Instead of configuring the environments, deploying the code, configuring the database, and running and debugging system tests, you need to create post-deployment automated tests that can be run after the environments are configured to make sure they are correct server by server. Do the same thing for the deployment and database. Then after you have proven that those steps executed correctly, you can run the system tests to find any code issues that were not caught earlier in the DP. This structured DevOps approach really helps to streamline the

triage process during code deployment and helps localize hard to find intermittent issues that only happen when a system test happens to hit the one server where the issue exists.

Making these deployments into production work smoothly requires these technical changes, but mostly it requires everyone in the DP working together to optimize the system. This is why the DP is an essential part of DevOps transformations. If Operations continually sees issues during deployment, they need to work to design feedback mechanisms upstream in the DP so the issues are found and fixed during the testing process. If there are infrastructure issues found during deployment, Operation teams need to work with the Development teams to understand why the infrastructure as code approaches did not find and resolve these issues earlier in the DP. Additionally, the Operations team should be working with the test organization to ensure post-deployment tests are created to improve the efficiency and effectiveness of the triage process. These are all very different ways of working that these teams need to embrace over time if the DevOps transformation is going to be successful.

Operation and Monitoring

The final step is operating and monitoring the code to make sure it is working as expected in production. The primary metrics to monitor here are:

- *issues found in production*
- *time to restore service*

Some organizations are so busy fighting issues in production that they are not able to focus on creating new capabilities. Addressing production quality issues can be the biggest challenge for these organizations. In these situations, it is important to shift the discovery of these issues to earlier in the pipeline. The operational organization needs to work with the development organization to ensure their concerns and issues are being tested for and addressed earlier in the pipeline. This includes adding tests to address their concerns and

adding monitoring that is catching issues in production to the test environments. As discussed in the release section, it also requires getting common tools and scripts for environments, deployments, and databases across the entire DP.

Implementing all these changes can help ensure you are catching most issues before launching into production. It does not necessarily help with the IE8 issue with Spanish localization discussed in Chapter 2. In that case, it would have just been too costly and time consuming to test every browser in ever localization for every test case. Instead, the other significant change that website or SaaS type organizations that have complete control over their deployment processes tend to implement is to separate deployment from release by using approaches like feature toggles and canary releases. This enables new versions of the system to be released into production without new features being accessible to the customer, a pattern known as "dark launching." This is done due to the realization that no matter how much you invest in testing, you still might not find everything. Additionally, the push to find everything can drive the testing cost and cycle times out of control. Instead these organizations use a combination of automated testing in their DP and canary releases in production. Once the feature makes it through their DP, instead of releasing it to everyone at once, they do a canary release by giving access to a small percentage of customers and monitoring the performance to see if it is behaving as expected before releasing it to the entire customer base. This is not a license to avoid testing earlier in the pipeline, but it does enable organizations to limit the impact on the business from unforeseen issues while also taking a pragmatic approach to their automated testing.

Summary

This simple construct of a DP with a single developer does a good job of introducing the concepts and shows how the DevOps changes can help to improve flow. The metrics are also very useful for targeting where to start improving the pipeline. It is important to look across all the metrics in the DP to ensure you start this work with the

bottleneck and/or the biggest source of waste because transforming your development and deployment processes is going to take some time, and you want to start seeing the benefits of these changes as soon as possible. This can only occur if you start by focusing on the biggest issues for your organization. The metrics are intended to help identify these bottlenecks and waste in order to gain a common understanding of the issues across your organization so you can get everyone aligned on investing in the improvements that will add the most value out of the gate.