O'REILLY®

Continuous Deployment

Enable Faster Feedback, Safer Releases, and More Reliable Software

Valentina Servile

Foreword by David Farley

Continuous Deployment

Enable Faster Feedback, Safer Releases, and More Reliable Software

This excerpt contains Chapter 1. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

Valentina Servile

Foreword by David Farley

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Continuous Deployment

by Valentina Servile

Copyright © 2024 Valentina Servile. All rights reserved.

Printed in the United States of America.

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

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

Acquisitions Editor: Louise Corrigan
Development Editor: Shira Evans
Production Editor: Jonathon Owen
Copyeditor: Audrey Doyle
Proofreader: Helena Stirling
Indexer: nSight, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

August 2024: First Edition

Revision History for the First Edition

2024-07-24: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098146726* for release details.

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

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-14672-6

[LSI]

Table of Contents

1. Continuous Deployment. 5
Deploying Every Few Months or Years 5
Deploying Every Few Days 6
Deploying Continuously 8
eXtreme Programming 9
If It Hurts, Do It More Often 10
DevOps 12
The Barrier Between Dev and Ops 12
Joining Dev and Ops 13
Automation, Automation, Automation 14
Continuous Integration 15
Continuous Delivery 17
A Final Gate to Production 19
One Step Further: Continuous Deployment 20
Implementation 24
Implications 25
Is It Dangerous? 26
Summary 27

CHAPTER 1

Continuous Deployment

For as long as software engineering has been a discipline, a great deal of care and attention have been given to application code and its architecture. All manner of paradigms, programming languages, and architectural patterns have originated to ensure that code is both well-organized in developers' editors and running efficiently later in production. It took more than half a century to collectively realize that we hadn't given enough thought to what happens in between.

Deploying Every Few Months or Years

Before the early 2000s, the average software product's path to production was an error-prone and clunky journey full of repetitive manual tasks. On this journey, changes from individual contributors were often integrated with delays, artifacts were built by hand, configurations and dependencies were tweaked outside of version control, poorly documented deployment steps were executed in meticulous sequences… and let's not forget testing, which was performed painstakingly by hand for every new version. As a result, release life cycles could span months or even *years*. Figure 1-1 is a fact-based depiction of those times.

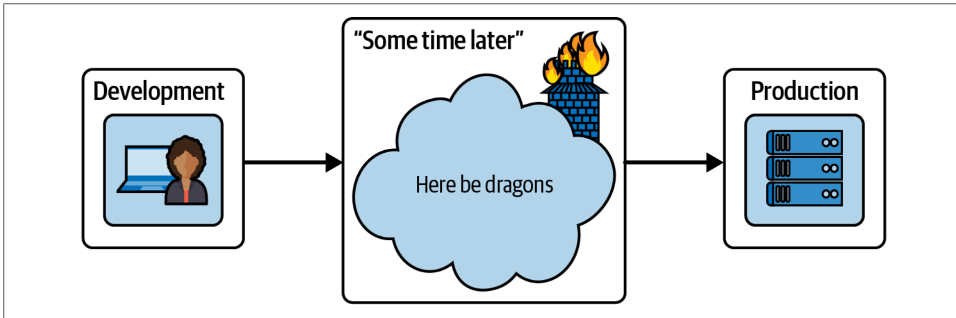

Figure 1-1. The typical path to production before the early 2000s

Such a lengthy path to production meant that all the up-front engineering investment in software design would not pay off until much later. Often, by the time new code finally found its way to users (if it ever did), the original requirements would have already changed—or, sadly, the market would have moved on, thus making the code irrelevant.

Fortunately, that has changed. Over the past two decades, businesses and organizations have become increasingly accustomed to shipping software rapidly in order to drive their operations without sacrificing reliability. Even in a (highly) hypothetical scenario where a company can brag about a flawless codebase and excellent product, a big-bang release every year or two is no longer a sustainable strategy for keeping up with market demands. That is why, in the past couple of decades, we have seen increasing attention being given to shortening the journey from "code committed" to "code running in front of users." The focus is no longer just on writing good-quality code, but also on ensuring that it is swiftly and painlessly released to production. After all, production is the only environment where code can repay the debt it has incurred from having been written in the first place.

Deploying Every Few Days

Companies have already been making their software delivery life cycles shorter and shorter, and they have achieved this mainly through the use of automation. The rationale behind automation's benefit is no mystery. Any human task or human decision represents a bottleneck in the fast-paced world of software development. Both our typing and thinking are several orders of magnitude slower than the pace at which code is executed—and both are vastly more error-prone. Therefore, major components of the path to production that used to be performed manually, such as building artifacts, testing, and deployment, have instead been automated through various tools over time (see Figure 1-2).

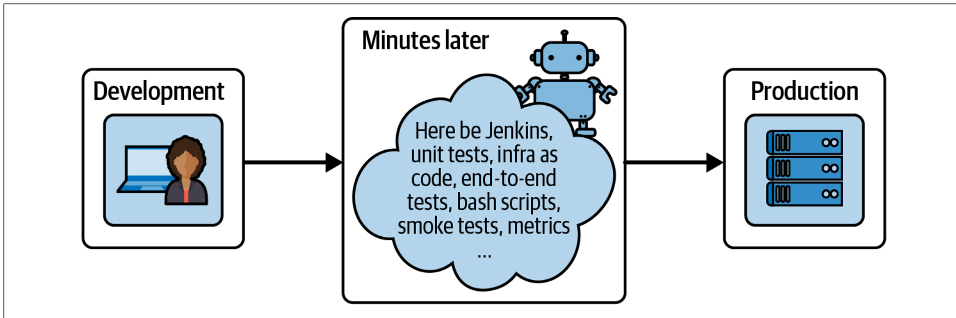

Figure 1-2. The typical path to production today

However, reducing the takeover of automation to merely "some tooling" would be doing a disservice to my readers, as the tooling is just one tangible result of something much more interesting: a set of schools of thought that changed the paradigm from writing code to *delivering software*. Practices such as eXtreme Programming (XP), DevOps, continuous integration (CI), and continuous delivery (CD), all now widely adopted, championed this shift in focus. If you haven't heard of them, I encourage you to familiarize yourself with them a bit more before diving into this book.[1] Continuous deployment stands on the shoulders of giants after all, and those are the giants' names.

XP, DevOps, CI, and CD introduced automation and repeatability into the path to production, from the moment code is checked in to the moment it is deployed, reducing manual intervention to a minimum. This automation provided the much-needed technical baseline to achieve speed without compromising safety. What followed was also a challenge for software engineers to cultivate an even broader set of skills around scripting and infrastructure, and to take ownership of their work on a level that far exceeds seeing their code run on a laptop. This mindset shift isn't constrained to only technical folks: these practices have also challenged organizations to rethink their overall culture (and structure) around the manufacture of custom software. Before them, huge batches of work were handed over from one segregated department to the next, with little to no communication between them. After them, we started seeing the emergence of cross-functional teams that are able to take care of their products from end to end.

1 Paul M. Duvall, *Continuous Integration: Improving Software Quality and Reducing Risk* (Boston: Addison-Wesley, 2007). Jez Humble and David Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* (Boston: Addison-Wesley, 2010). Kent Beck, *Extreme Programming Explained: Embrace Change* (Boston: Addison-Wesley, 2004). Gene Kim et al., *The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations* (Portland, OR: IT Revolution Press, 2016).

These movements have already produced results that used to be unthinkable: making software release cycles more reliable and dramatically faster. As a result, more than two-thirds of companies are now able to deploy to production *between once per day and once per month*.[2]

Deploying Continuously

Continuous deployment is another step forward along the progression of engineering excellence that has seen automation taking over software's path to production. In a sense, continuous deployment represents its logical culmination. While its predecessors automated some or even *most* of the code's journey to users, continuous deployment automates it all. Even the decision to perform the final production deployment is made by a pipeline agent, not by a person.

In a nutshell, continuous deployment means that a commit being pushed or merged into the main branch always results in a production deployment (provided all quality gates are green). For the first time, the journey from "code committed" to "code running in front of users" encounters no manual intervention whatsoever, and it is completed at the granularity of every commit. By removing even the very last human bottleneck, continuous deployment further reduces the software feedback loop from days or hours to mere minutes.

Such a dramatic acceleration has important consequences for engineers, who work with the knowledge that the tiniest increment of code will be subjected to full production-quality feedback practically immediately. With continuous deployment, tasks are no longer "dev-done, pending production deployment is successful later"; they are actually and demonstrably done. Production deployments themselves become trivial in virtue of their frequency, since the delta of changes is just a few lines as opposed to days' or weeks' worth of accumulated changes, greatly reducing the risks associated with each deployment. (I will throw more light on the relationship between deployment size and deployment frequency in Chapter 2.)

But as is the case with most engineering practices, the real paradigm shift in continuous deployment is best evaluated from the point of view of the businesses that adopt it. If production code increments can be maximally granular and lightning fast, product increments can be too. Small (or big) product decisions can be made and reversed much more easily, and made at the last possible moment—which is also the most

2 See *https://oreil.ly/ZJS1G* to read the DORA "Accelerate State of DevOps Report 2023" in full.

responsible way to make such decisions.[3] Sudden surprises in user behavior or requirements can be addressed immediately, with no pending deployments clogging the path to production and complicating last-minute changes. Ordinary releases themselves are broken down into slices so small that it is possible to experiment and gauge user behavior with precision, if that is what one wishes to do. Such a process takes the "responding to change" value of Agile to the extreme, and offers the narrowest possible feedback loops within the paradigm of iterative development.

In my professional experience, continuous deployment has resulted in an improvement to all the metrics that mattered to my teams when delivering software: the time it took to deliver any given change, the number of changes we could deliver overall, the rate of defects, and the speed at which we recovered from them. In this book, I want to share that experience because I believe the conversation about the journey from "code committed" to "code running in production" is still very much ongoing. I hope this book will represent a positive contribution to this: placing continuous deployment on par with all those other well-beloved practices.

So how exactly does continuous deployment work? What else does it contribute to code's path to production that hasn't been achieved already? How does it differ from its very close relative, continuous delivery? And why does that difference matter?

To answer these questions, we need to go back to the origins of this automation movement and spend some time understanding the practices that came before it. Only by understanding the principles guiding its predecessors can we appreciate how continuous deployment builds on them.

Let's start chronologically: with the introduction of *eXtreme Programming*.

eXtreme Programming

Tasks such as testing, reviewing, integrating, and deploying are central to code's path to production, but they can also be painful and awkward to execute. As a result, they used to be relegated to the very end of long iterations, completed infrequently, and completed mostly by hand.

The change in discourse around these tasks picked up between the late 1990s and early 2000s with the popularization of eXtreme Programming (XP), which brought these noncoding activities at the margins of software development to the center of the conversation. XP is an Agile software methodology that emphasizes doing the most important parts of software development (which are not necessarily *writing* software)

3 The last responsible moment (LRM) is the strategy of delaying a decision until the moment when the cost of not making the decision is greater than the cost of making it, as explained by Mary and Tom Poppendieck in their book *Lean Software Development: An Agile Toolkit* (Addison-Wesley, 2003). This strategy helps you make decisions only when the maximum amount of information is available.

as often as possible. As Kent Beck writes in *Extreme Programming Explained*, "Take everything valuable about software engineering and turn the dials to 10."[4] This process involves practices such as pair programming, where two programmers work together at one workstation (thus reviewing code continuously); test-driven development, which involves writing tests before the code (testing continuously); and continuous integration, where code changes are integrated in the main branch as frequently as possible (this one even has "continuous" in the name).

XP was among the first movements to challenge the old-fashioned behavior of delaying painful steps, and it suggested instead that increasing the frequency of these tasks (doing them continuously) will in turn reduce their difficulty. It might seem counterintuitive, but the motto of XP was the first principle that enabled the automation of code's journey to production: "If it hurts, do it more often."

If It Hurts, Do It More Often

When I started out, I thought that if reviewing, integrating, testing, and deploying were painful, we would surely want to put them off as much as possible. Why on earth would we want to do them more often than we needed to? We could write our code in our little walled gardens (or feature branches) for weeks and weeks, and when the merge-and-release hell could not be postponed any longer, we could put our heads down for a few days and just get them over with. Then we could forget all about the ordeal and return to the peace and quiet of our personal code gardens. That is, until somebody asked our team to show another working version of the software, including all the new features we had been working on individually. Then, we would have to start all over again. *If only those annoying stakeholders stopped interrupting us with their demos and releases*, I thought. *Then, we developers could get the* real *work done around here!*

I started to grasp what "If it hurts, do it more often" really meant only after working in teams where we had to integrate and release on different schedules. Curiously, the shorter the cadence was, the less work our team had to do in order to get our software ready, and the less rework came back to our desks. "If it hurts, do it more often" is not about feeling more pain from integration and deployments but about how increasing their frequency enables us to feel *less* pain. And not just less pain in each release, but less pain overall.

Smaller, less-painful batches

Doing the most-painful activities, such as merging, testing, and deploying, more often means that smaller and smaller amounts of code need to go through that

4 Beck, *Extreme Programming Explained*, p. 127.

activity. That is because commits don't get a chance to accumulate while they wait to be processed. The more often we integrate, test, and deploy our software, the less software we need to integrate, test, and deploy each time.

By integrating often, change deltas get small enough to make merges straightforward. By deploying often, bugs that could have taken days of investigation can, within hours, be traced back to the change that introduced them; after all, the lines of code to sift through are so much fewer.

Incentive for automation

Doing things more often (and in smaller batches) is also a strong incentive for developers to streamline repetitive and risky processes. Automating a task—however nightmarish—that used to happen once every blue moon didn't make much sense before short iterations became the norm. But automating a daily sequence of error-prone steps certainly did. And so automation started to take over, and now we have pipelines automatically polling our latest code changes from version control, building artifacts from it, running all sorts of automated tests, and even performing deployments all on their own. We tend to automate these tasks when doing them more often highlights slowness and inefficiencies and frankly makes them just a lot of work. One of the best virtues of us programmers is our laziness.

As a result, the integrate, test, and release phases that we used to dread have become much less painful—basically, routine for a lot of companies. This didn't happen *despite* markets demanding we do these things more often, but *because* companies started doing these things more often.

Reduced pain and increased frequency have also allowed our businesses to gather feedback on new initiatives much earlier and helps reduce the ultimate waste: building things that users don't want or need.

Applying this idea of "doing painful things more often, in smaller batches, and automating as much as possible" has radically simplified the most painful parts of shipping software. But as we discussed, these improvements didn't happen all at once. Different practices took over from XP's early good instincts to play a big role in the following years. Following the "If it hurts, do it more often" principle, software's path to production was later automated one chunk at a time: starting from building artifacts and ending up at deployments to production.

For example, it could be said that the well-known practices of CI and CD are respectively rooted in the pain of integrating our code with other people's and the even worse pain of delivering production-ready software. Some might even say that the practice of DevOps is rooted in the pain of miscommunication and friction between traditional Dev and Ops departments.

Few people might quote XP nowadays, but the mindset that it sowed continues to guide teams and organizations as they shrink their feedback loops and go to market more and more quickly. Let's go through a bit more history to understand what happened following its introduction.

DevOps

DevOps is not a job title (sorry, recruiters), but the name of a movement that emerged as a response to the dysfunctional separation between the activities of writing and running software. It emerged between 2007 and 2008, and in 2009 Patrick Debois founded the first DevOpsDays conference.[5] The movement continued with the publishing of two key pieces of literature: *The Phoenix Project*[6] and later, *The DevOps Handbook*.[7]

DevOps aims to break down the traditional barrier between software development and IT operations, which used to be distinct and siloed departments. In other words, developers used to be focused solely on writing code, while operations was responsible for deploying it and maintaining it in production.

The DevOps movement sought to change that, as it identified that separation as the culprit for much of the pain associated with releasing and operating software at the time. Instead, it encouraged a culture of collaboration and integration between the two functions, with the aim of establishing a path of continuous learning from each other. On a technical level, this also encouraged continuous learning in terms of lessons learned in production being fed back into the development process.

Let's take a deeper look at those dysfunctions to understand why.

The Barrier Between Dev and Ops

Before DevOps, most development and operations teams worked in isolation from each other—sometimes physical isolation, with departments on different floors or even in different buildings. This would quickly lead to a lack of understanding of each other's work and priorities, and difficult handovers between departments. Of course, miscommunication would only get worse when the latest deployment wouldn't work as expected (which was often), leading to a lot of finger-pointing and a general "throwing stuff over the wall" attitude.

5 Steve Mezak, "The Origin of DevOps: What's in a Name?", DevOps.com, *https://oreil.ly/TCzso*.

6 Gene Kim et al., *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win* (Portland, OR: IT Revolution Press, 2014).

7 Kim et al., *The DevOps Handbook*.

The separation also led to *knowledge silos*, which hindered the ability to address problems effectively (especially the ones that sat at the boundary between application code and infrastructure). Devs had limited visibility into how their code performed in production, while Ops had limited insight into the codebase and its dependencies and configuration. This lack of visibility made it much harder to diagnose issues, resulting in longer resolution times and overall increased downtime. It also made software more likely to develop those issues in the first place, since it wasn't written with production conditions or operability in mind.

However, and even more importantly, this "handover" process between departments was a major culprit in the slower delivery cycles at the time. A code handoff between departments for the purposes of deployment would, of course, lead to lengthier deployment times. Lengthier deployment times would then cause delays in releasing new features and an inability to quickly respond to changing requirements or feedback.

Addressing all of these dysfunctions would only be possible by removing the organizational problem at the heart of them: the separation between departments.

Joining Dev and Ops

Because of these issues, the DevOps movement advocates bridging the gap between the two functions. It should be no surprise that the primary tenet of the movement is focused on communication, collaboration, and a fundamental organizational reshuffling.

DevOps is all about avoiding handovers and ensuring shared ownership of the code, its deployment process, and its permanence in production. Nowadays, a "true DevOps team" is cross-functional. It is a team that makes both code and infrastructure changes, deploys them, and supports the resultant systems in production; as Amazon CTO Werner Vogels famously said, "You build it, you run it." And it doesn't stop at infrastructure. DevOps advocates for all functions required to evolve and maintain a product to be embedded into the team (e.g., quality assurance, security). This doesn't mean that specialized engineers are a thing of the past, of course. It means that each team can, and should, take accountability of its application's tech stack from top to bottom. To quote from *The DevOps Handbook*:

> Imagine a world where product owners, Development, QA, IT Operations, and Infosec work together, not only to help each other, but also to ensure that the overall organization succeeds. By working toward a common goal, they enable the fast flow of planned work into production (e.g., performing tens, hundreds, or even thousands of code deploys per day), while achieving world-class stability, reliability, availability, and security. In this world, cross-functional teams rigorously test their hypotheses of which features will most delight users and advance the organizational goals. They care not just about implementing user features, but also actively ensure their work flows smoothly and frequently through the entire value stream without causing chaos and disruption to IT Operations or any other internal or external customer. [...] By adding the

> expertise of QA, IT Operations, and Infosec into delivery teams and automated self-service tools and platforms, teams are able to use that expertise in their daily work without being dependent on other teams.[8]

This shift from siloed departments to cross-functional teams was fundamental. Collaboration early and often in the software life cycle improves the quality of the software itself because all the necessary perspectives are considered early in the development phase, reducing issues and rework later on. For example, software becomes more operable in production because developers finally benefit (or suffer) from the operability consequences of the decisions they make. I consider this to be a perfect example of the "It hurts [communication between Dev and Ops], so we do it more often" principle in practice.

Collaboration and cross-functional teams are also what unlocked the automation of software's path to production, which is the second tenet of the DevOps movement and the focus of this chapter. After all, automation of the path to production would not have been possible if its ownership had been scattered across multiple departments.

Automation, Automation, Automation

The other core tenet of the DevOps movement is indeed its explicit focus on automation. Automating repetitive and manual tasks such as infrastructure provisioning, configuration management, testing, and deployment has been a key interest of DevOps for all the reasons we already discussed: it is faster, reduces risk, and allows teams to streamline otherwise clunky and error-prone processes.

While automation had been popular before, DevOps is the movement that has put the most emphasis on it, elevating it to a core principle and a first-class citizen instead of a pleasant side effect of other practices.

The rise of cloud computing was also conveniently timed around the emergence of DevOps, and it made for an excellent technical baseline to support this principle.

Cloud computing gave developers a much-needed abstraction layer on top of physical infrastructure, and it diminished the need for in-depth expertise in the areas of server and network maintenance. It also allowed teams to quickly tear down and rebuild infrastructure, which has tended to make it more immutable and predictable. To understand why, I will rely on a popular distinction in the DevOps community: the one between "pet" and "cattle" servers. Servers in on-premises data centers are often treated as "pets," whereas servers in the cloud are more analogous to "cattle." Pets are heavily customized servers that cannot be turned off (and like real pets, they usually belong to *someone*, which goes against the shared ownership principle). On the other

8 Kim et al., *The DevOps Handbook*, Introduction.

hand, cattle servers are disposable: they can be deleted and rebuilt from scratch in case of failures, losing any customization in the process. This means immutability becomes a requirement and one-off manual changes become an antipattern, which is a great baseline for automation.

The abstraction layer given by cloud computing has since evolved even further, with *infrastructure as code* (IaC) tooling becoming widespread. IaC has allowed teams to represent the infrastructure required to run their software as version-controlled "executable" files. This has allowed for even further automation, and it has made infrastructure just as easily versionable and self-documenting as application code, not to mention more easily manageable by developers.

The ability to both abstract away the physical infrastructure and represent it within its own codebase is what unlocked deployment capabilities for automated pipelines, which are fundamental to the practice of continuous integration and continuous delivery. Those are the practices we will talk about next.

Continuous Integration

The next fundamental practice for the automation of software delivery was *continuous integration* (CI), which radically changed the initial steps of the path to production. It was popularized in 2007 by Paul Duvall in *Continuous Integration*. Actually, continuous integration was already described as a practice by XP, but since the supporting practices that surround it took a while to develop and the Duvall book is the most complete account of it, I'm inserting it into our story here. Since then, CI tools have proliferated and have become mainstream.

Due to the abundance of tools and their popularity, continuous integration has suffered from semantic diffusion[9] and has become synonymous with automated pipeline tools over the years. But at its core, CI is not an installation of Jenkins, CodeShip, or Travis. It is the practice of integrating developer changes as frequently as possible in the same version-controlled branch.

Integrating big chunks of changes is painful and error-prone (it hurts), so it can be improved by doing it continuously (we do it more often). In short, CI is about ensuring that the amount of code to be merged and verified is reduced to a minimum.

CI can be achieved by adding code changes to "trunk" (master, or main) as often as possible. *Trunk* is where the latest version of everyone's code resides, so it's the only place where true *integration* can happen. In practice, this is best achieved with *trunk-based development* (TBD), where code is always pushed directly into the main

9 Semantic diffusion is the phenomenon in which a term is coined with a very specific definition, but then gets spread through the wider community in a way that weakens that definition.

branch. TBD means forsaking feature branches, which are the very definition of "walled gardens" of code, where commits can pile up and integration can be endlessly delayed.

So, is it impossible to do CI with branches? We will explore this topic in more depth in Chapter 3, but for now I can say this: it's not strictly *impossible*, but it is certainly much more difficult. Some teams do achieve successful continuous integration by using short-lived branches and/or pull requests (PRs), but they do so by still integrating into trunk as often as possible—ideally, at least once a day.[10] So, while it is possible, branches and PRs make it difficult to do the right thing (remembering to integrate often) and easy to do the wrong thing (adding "just" another commit without integrating). TBD does the opposite.

Now that we have talked about integration, let's talk about the better-known aspect of the CI methodology: the automated build pipeline. You can see it in Figure 1-3. This is how it works: once code is merged or committed to trunk, a centralized server shared by the whole team will detect a change in the version control system. When it notices the change, it builds the code into an artifact. An *artifact* is the final "bundle" containing all the code, dependencies, and configuration necessary for the new application version to work. For example, an artifact could be a compiled binary, an archive, a *.jar* or *.war* file, or even better, a container image. Once that artifact is created, the pipeline verifies its correctness through automated tests and code inspection tools. The shared build pipeline then gives feedback to the team through information radiators, and notifies them whenever any of the checks have failed.

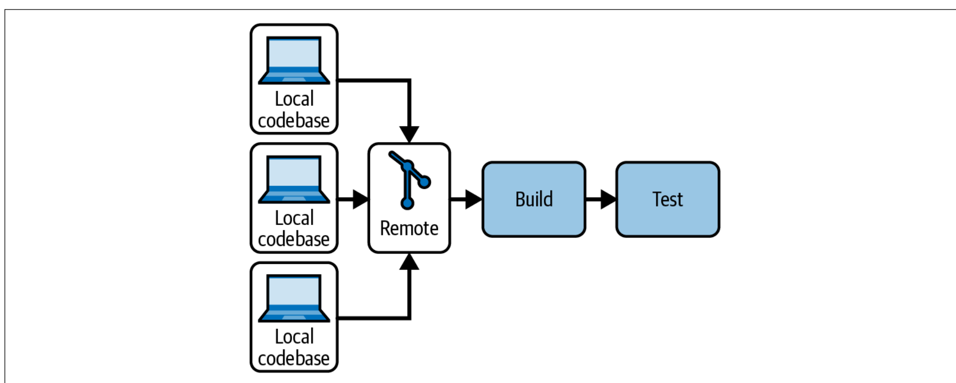

Figure 1-3. The CI automated pipeline

This automation around the pipeline, coupled with integrating early and often, highly improves the granularity of the building and testing phases. Whereas before they

10 Martin Fowler writes about continuous integration at *https://oreil.ly/SqElv*.

were performed once in a while, now they are triggered for every commit, shortening the integrate-build-test feedback loop and making defects much easier to spot.

The automated build pipeline is indeed a powerful concept, but CI is more than a tool. It's a set of practices and mutual agreements that operate within a team. Integrating often and keeping the shared pipeline green are the crucial principles of CI, as the pipeline state represents the fundamental ability to have a proven working version of the software. If we have a proven, latest working version of our code, being ready to deploy it on short notice isn't much of a stretch anymore.

If the pipeline is red, or if we haven't merged in a long time, there is no latest artifact and nothing new to deploy: it's as if the new code were not there at all and the team hadn't made any progress since the latest green build. As David Farley said to me while reviewing this book, "That's because in practical terms, they haven't!"

Having this centralized source of truth and validation has radically changed developers' ways of working, shifting the definition of *done* from "Well, it works on my machine" to "It works once it's integrated, built, and *proven* to work." Merging, building, and testing were previously painful tasks that have now become an ordinary occurrence, removing the need for any last-minute scrambling to get something ready. And stakeholders couldn't be more thankful for it.

As I mentioned at the beginning of the chapter, before the concept of CI came along, there was little engineering around the process for code to reach its users once it had been written. Continuous integration was the first practice to change that, and the concept of a centralized pipeline has been the basis of all automation that followed.

Continuous Delivery

Shortly after continuous integration came the concept of *continuous delivery* (CD), which can be seen as the obvious next step in the automation of the path to production. It was made popular in 2010 by Jez Humble and David Farley in *Continuous Delivery*, and since then it has enjoyed a similar level of popularity as continuous integration. It has become the latest de facto standard for engineering practices and the one most companies strive to implement today. This is made evident by the fact that most popular automated pipeline tools now feature all the capabilities required to perform CD as well as CI.

Indeed, the technical baseline for this practice requires extending the functionality of the automated pipeline (Figure 1-4). The pipeline itself still builds and validates the latest version of the code, but CD also states that it should be able to deploy to any environment in an automated fashion—at any time. This can be achieved through IaC and provisioning tools (another gift of the DevOps movement), which guarantee the automation and repeatability of deployments.

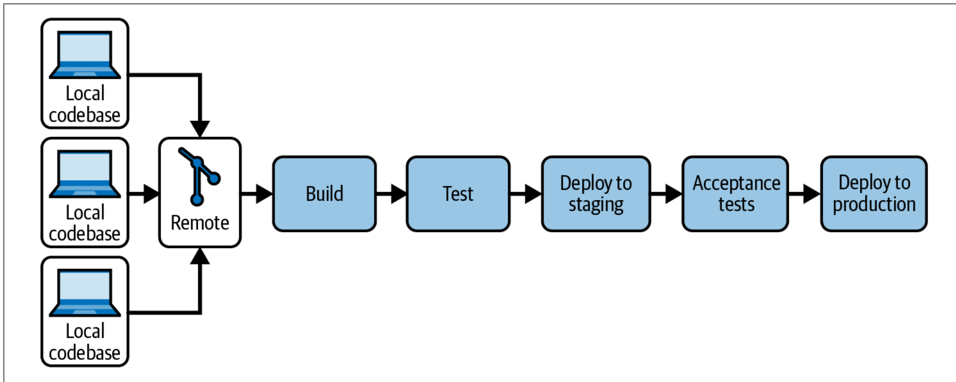

Figure 1-4. The CD automated pipeline

In the most common implementation of CD, every build will be automatically promoted and deployed to a staging or UAT environment, or even several of them. Once deployments are automated, sophisticated automated tests can then be run against those prod-like environments. As a result, each code increment that makes it to the end of the pipeline has such a high likelihood of correctness that it can be considered a release candidate. The deployment to production of this release candidate can then be triggered on demand, usually through the same pipeline tool.

With CD, the automated pipeline owned by the development team becomes the tool that covers the application's entire path to production. In doing so, it finally bridges the gap between traditional Dev and Ops functions, allowing one team to fully own both development and deployments.

However, just like CI, CD doesn't represent only a tool. First and foremost, it represents a collection of ways of working to leverage the tool successfully. In the case of CD, those ways of working are centered on one simple concept: ensuring that the codebase is always in a deployable state (treat every commit as a release candidate) so that the automated pipeline can be leveraged to deploy to production at any moment. This allows teams to dramatically increase their deployment frequency. A higher deployment frequency leads to fewer and fewer lines of code between one deployment and the next, which lowers the average deployment risk. Once again, this is the "If it hurts, do it more often" principle in action.

The shorter feedback loops introduced by this practice also bring to the table something else that is far more valuable than reducing the annoyance of deployments. With CD, showing the latest version of our software to stakeholders isn't nearly as much of an inconvenience as it was before. The most recent version of the software is likely sitting on a freshly deployed pre-prod instance at all times, having been delivered there by the latest pipeline run. In most cases, all we have to do is open a new tab in our browser to get a working demo in a real environment. This used to be

unthinkable in the days of manual deployments, especially with the added communication overhead due to having separate Dev and Ops teams.

With CD, stakeholders can see the software quickly, and releasing it to end users is just another step away. Deploying to production doesn't have to be scheduled months in advance; it can be easily done several times a week. Requirements can be validated much more quickly than before, by testers, stakeholders, and most importantly, the market. It turns out that when we deploy more often, we enable our organization to get feedback—and adjust to the market more often too. In many cases, this results in building less software, as early feedback causes product owners to realize a feature or an entire system got to "good enough" earlier than expected.

If continuous delivery has established anything, it's that "the real work around here" for us developers was never to produce endless code in a walled garden, but to release working software early and often. In only this way can we bring value to users as quickly as possible.

In short, the evolution from CI to CD can be framed in terms of what is being automated: CI automates the determination of "Does my change look safe and self-contained?" while CD automates the determination of "Is my change deployable?" Later, we will see how CD automates the question, "Shall I deploy *now*?"

A Final Gate to Production

Most companies implement continuous delivery in such a way that deployments happen without human intervention in at least some preproduction environments for every build. However, the same automation does not carry changes all the way to production. To cover that last step, a human still has to press a button in the pipeline tool to deploy the release candidate (Figure 1-5). The presence of that manual "button" is the main differentiator between continuous delivery and continuous deployment.

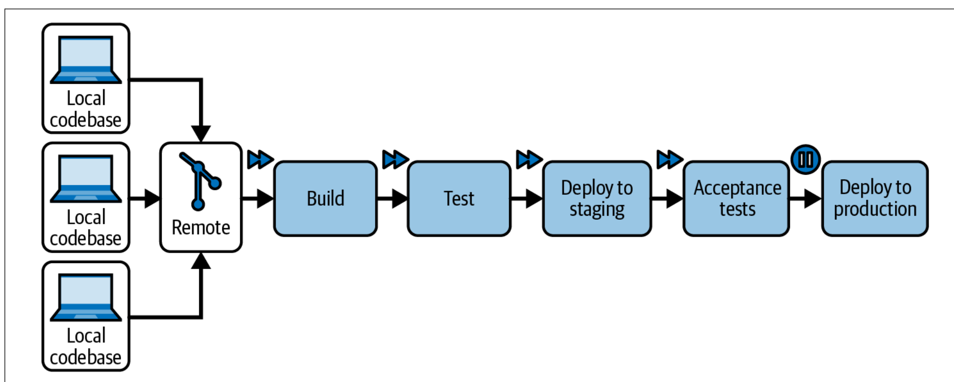

Figure 1-5. The final gate to production

Even with a final manual gate, many more phases of the path to production are streamlined: deployments to preproduction and tests at higher levels of abstraction can be executed for every increment of code rather than for weeks' worth of accumulated changes. Ninety-nine percent of the path to production is completely automated. It is close to perfect.

However, this implementation of continuous delivery still has one remaining bottleneck in the overall process of getting code in front of users: the last human's manual approval to perform that final deployment. This 1% might look insignificant, but it does make a difference.

Teams might handle production deployments in different ways, ranging from ad hoc deployments when there are enough substantial changes, to sophisticated schedules where deployments happen on a fixed cadence. By the time manual intervention happens, however, a batch of commits might have accumulated in preproduction, and they will usually be deployed all at once—increasing deployment risk, delaying feedback, and slowing down debugging if (when) something goes wrong.

The bigger the deployment is, the more oversight is required, and the more potential defects lurk in the growing number of lines of code waiting to be rolled out. Also, the more changes that are made at once, the more code we must dig through to diagnose the cause of post-deployment issues.

If the process of deploying to production requires lengthy manual verification (e.g., extensive manual testing in staging or UAT), it is not really a continuous process, and so it might not even strictly qualify as continuous delivery. Manual intervention defers the point of release even further and exacerbates this problem, making each deployment bigger and more complex.

In addition, just like with TBD versus branches, a gated production environment makes it difficult to do the right thing (deploy as often as possible) and easy to do the wrong thing (add "just another commit" before deployment).

Even though most companies are enjoying great benefits from continuous delivery, they are still not deploying to production as often as they could; that is, with the granularity of every commit.[11]

One Step Further: Continuous Deployment

Now that we have learned about the principles of XP, DevOps, CI, and CD, we can finally understand the thinking behind continuous deployment.

11 Aficionados of Lean manufacturing will know this as "one-piece flow." See Chapter 2 for more details.

A couple of decades after the discourse around the path to production started changing, the DORA State of DevOps research program, led by Nicole Forsgren, came along to validate XP's earlier instincts about doing the most painful parts of software frequently.[12] This program is the longest-running academic investigation into software delivery practices so far. Through six years of research, it has confirmed that a shorter lead time for changes and a higher frequency of production deployments are reliable predictors of high performance in software development teams (and their organizations). It is also worth mentioning, from a selfish point of view, that the DORA research has found a strong correlation between those metrics and better quality of life for engineers, with overall higher job satisfaction and less burnout. These findings apply to both commercial and noncommercial organizations.

Encouraged by the DORA findings, teams that already follow the values of XP, DevOps, CI, and CD are now trying to make their deployments happen more and more frequently and with less and less gatekeeping. Engineering practices that enable deploying to production as often as possible have become a key focus of organizations looking to reduce their time to market and their ability to respond to change.

Therefore, it is only natural that some teams might be thinking of going one step further with continuous delivery and removing the final manual barrier to production: automating the last step of their automated pipeline, and therefore enabling continuous deployment (shown in Figure 1-6).

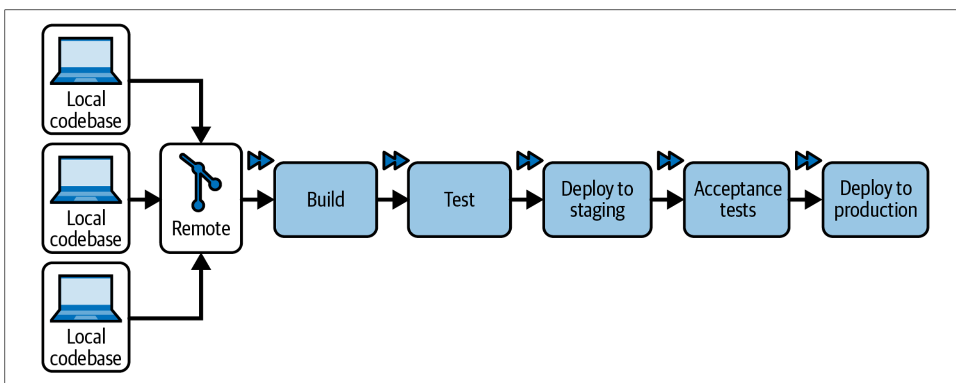

Figure 1-6. The continuous deployment automated pipeline

The term *continuous deployment* was coined by Timothy Fitz in his 2009 article of the same name. The concept hasn't been included in a lot of literature since then, but it has gained traction and has been successfully adopted by companies such as Facebook, Google, and many more. Because continuous deployment is still relatively

12 You can download the DORA "Accelerate State of DevOps Report 2023" at *https://oreil.ly/epiI8*.

undocumented, there is no official data showing exactly how many companies have adopted it overall (most studies focus more generally on continuous *delivery*, not specifically on continuous *deployment*). That is why I have collected several case studies in this book: so that you can have an idea of how a good sample of companies make it work in their day-to-day in all sorts of industries: from retail to banking to automotive and more.

Continuous deployment can be understood as a specific implementation of continuous delivery, acting within its framework of ensuring that every code increment is production ready. Indeed, continuous deployment takes the "production-ready" principle quite literally. Its defining characteristic is that the decision to deploy code to production is fully automated by the pipeline, with no manual intervention required.

Every commit going through the pipeline not only will be *considered* deployable to production, but will actually *be* deployed to production.

This means no more waiting around in a UAT or staging environment for manual exploratory testing, and no more human intervention to decide whether to send the changes live. The pipeline does it all, and humans can finally sit back and watch the automation do its job until the end.

We have already talked about how the "If it hurts, do it more often" principle of XP is reflected in the practices of DevOps, CI, and CD. All these years later, I consider the same principle to be the main motivation behind adopting continuous deployment to production as well. Perhaps continuous deployment is the practice that takes this concept to its very limit.

What has *ever* been more painful for any software team than deploying to production? Integration with the live versions of third-party services, high-traffic conditions, unforeseen user behaviors, and real user-generated data has historically wreaked havoc on even the best-laid-out release plans. And the consequences have been much worse than a bad merge day or a disappointing demo. Real users have been presented with broken applications, unusable features, and even the loss of their data. Deploying to production has led to business outages lasting hours or days for countless teams. As we will see in later chapters, the production environment is full of quirks that are hard or impossible to replicate on developer laptops. Even the fanciest, most sophisticated preproduction setups often aren't quite production-like enough to catch all those quirks.

Deploying to production is a scary ordeal, and when it goes wrong it can disappoint a lot of people. I have a hard time thinking of anything more painful for a software team than deploying to production after days or weeks of hard work, only to realize that their code had an unforeseen defect that is now being paraded in front of users and stakeholders. Most experienced developers know how mortifying that can be. Or

even worse, they might know the disappointment of working long hours to get the Big New Feature over the finish line, only to find out users are happily ignoring it.

Deploying to production can definitely hurt. Therefore (in true XP fashion), we should do it as often as possible; ideally, at every commit that is proven worthy by our best automated tests. Only then will it be possible to ease that pain and actually look forward to deploying multiple times a day instead of dreading it once a month. That's what continuous deployment is, and in a nutshell, that's what this book is about.

In my view, continuous deployment represents the culmination of the decades-long automation journey I have described in this chapter. With it, automation can finally cover the entire path to production from end to end and leave no manual work in the value stream of software.

With immediate deployments, the code commit life cycle goes into lockstep with the deployment life cycle. Each code increment in the codebase corresponds to an almost immediate code increment in production, so every commit must be production ready from its inception. Developers work within the narrowest possible feedback loop every day, and even every hour.

Indeed, every developer might easily oversee multiple production deployments while creating code for a single task. This makes them intimately familiar with the production environment and all its quirks, interdependencies, and performance constraints. Continuous deployment fully empowers developers to take responsibility for both the value *and* the destructive potential of their changes. Overall, this framework promotes production to a first-class citizen in the day-to-day cognitive effort of writing code. It already should be, but it is easy to forget about it when deployments happen "sometime later" or are overseen by someone else.

With a 1:1 correspondence between commits and deployments, the state of the codebase and its relationship to the state of production becomes straightforward: they are always one and the same. There is no need to switch to an older version of the code to debug live issues, no question about which version is out at the moment, and no need to revert undeployed changes if an urgent fix has to be rushed through. The tiniest of experiments can (and should) go live in a trivial amount of time: ideally measured in minutes—whether it is a new log line for debugging, or changing the color of a button under a feature toggle.

Overall, working in this fashion reduces the risk of production unpredictability by managing it in the smallest possible increments: within the granularity of every commit.

Implementation

The implementation of continuous deployment itself can be quite simple. Adapting an existing continuous delivery pipeline to support continuous deployment usually only requires a reconfiguration of the production deployment step.

If your software respects all of the following conditions:

- It is version controlled.
- It has automated test coverage.
- It has an automated pipeline.
- The automated pipeline runs on every new commit on trunk (the main branch).
- The automated pipeline is responsible for the entire path to production, from commit, to tests, to deploying to any environment (including production).
- The "trunk" version of the software is deployed to your main preproduction environments and, of course, to production.
- The automated pipeline runs in a reasonable amount of time (say, all steps combined should take less than one hour, discounting any pauses for manual checks).

After these conditions are satisfied, all you need to do is ensure the following: once a commit is pushed (or merged) to trunk and all quality gates have passed, the deployment to production happens immediately. There should be no manual steps, and no buttons waiting to be clicked by a carbon-based life form. If you have already adopted continuous delivery, this usually means deleting a "pause" instruction before production deployment in your pipeline tool of choice.

If there are any other required change approvers, or any earlier manual gates in between a push and a production deployment, they should also be automated. When any push or merge event to the main branch is able to trigger its own production deployment, that's when you know you are done.

How do you know when your automated quality gates and infrastructure are good enough to make this switch? If you are looking for an indication of when you might be "ready," you might enjoy reading Chapter 3.

Continuous Deployment to…Staging?

Some companies refer to their release process as continuous deployment, but with the caveat that it is only *continuous* until staging or UAT. After that, somebody needs to give a go-ahead to go to production, because it is deemed "too risky" to let it happen by automation only. I would suggest that continuously deploying to a preproduction environment is a natural part of practicing continuous delivery, but it is not really

continuous deployment. On this topic, I rather agree with the words of David Farley and Jez Humble, who began to explore the topic back in 2010:

> Of course it's not just continuous deployment (I can continuously deploy to UAT all I like: no big deal). The crucial point is that it is continuous deployment *to production*.
>
> —Farley and Humble, *Continuous Delivery*, p. 266

With continuous deployment, according to the original definition, *all* manual gates are removed from the automated CD pipeline. Each step triggers the next step in an automated fashion, and the culmination of all those steps is a production deployment. Any code change pushed to the main branch is a fully fledged release candidate. If proven correct by the automation, it *will* make it all the way to the users.

And that's it. There will be no more pipeline implementation details in this book, because this is really all there is to "implementing" continuous deployment.

A continuous deployment pipeline is certainly not laborious to build. To switch from continuous delivery to continuous deployment, Very Senior Developers don't have to spend weeks coding complicated scripts full of deployment steps, configuration of cloud permissions, infrastructure as code, and elaborate testing setups. In most cases, all they need to do is remove a step from their existing pipeline. Implementing continuous delivery from scratch might take months or years, while switching to continuous deployment often just requires a one-line change. As most teams these days have already invested in a continuous delivery process, it certainly looks like the switch might not be that hard for most companies.

So, what are we going to talk about in the rest of the book, then?

Implications

Continuous deployment could be dismissed as a trivial subcategory within continuous delivery, but this underestimates the radical simplicity of an automated pipeline that goes straight through from push to production.

The challenges of continuous deployment do not lie in its deceptively simple implementation. Rather, they lie in the adoption of all the practices that it depends on and the ones that it unlocks. It might be a one-line change, but it is a one-line change that (in my experience) represents a complete reimagining of the day-to-day software development process.

For example, these are just some of the questions that we'll address in the rest of this book, and that emerged for my teams only *after* the switch to continuous deployment:

- How do we hide unfinished code if we are deploying every commit?

- How do we ensure that *every single* commit is backward compatible?
- How do we avoid breaking contracts with other production services?
- How do we separate a deployment and a feature release?
- What is the effect of very frequent deployments on the stability of infrastructure?
- How does this change how the team works and its definition of "done"?
- What happens with preproduction environments?
- Is anything ever "dev-done" anymore if everything is in production?
- How can manual exploratory testing still be done when deployments are immediate?

A continuous deployment workflow is radically different from a traditional continuous delivery workflow with a gate to production. So, because we have already talked about how to "implement" continuous deployment, the rest of this book will cover its ramifications and the implementation of all of its supporting practices.

Is It Dangerous?

If your heart skipped a beat when you read about manual production approvals being removed, don't worry. You are far from alone.

When teams practice good continuous integration and use of short-lived feature branches or, even better, trunk-based development, they send multiple changes through the pipeline every day. Some teams might even send multiple changes every hour, all of which will end up in production one after the other. This streamlines the engineers' workflow considerably, but it is also a responsibility not to be taken lightly.

Each commit ending up in production requires developers to be extremely disciplined with all the principles introduced by continuous delivery. The pipeline, preproduction environments, and automated tests must be maintained to impeccable shape so that they can accurately discard bad changesets. For developers, treating every code increment as a potential deployment candidate also becomes mandatory. After all, each commit will become an actual deployment: they are not just "candidates" anymore.

Indeed, one might argue that the production readiness offered by continuous delivery is only theoretical without continuous deployment.

But as cool and shiny as this practice is, it does present some risks: because each change developers make goes immediately to production, it has the potential to affect a complex web of services. A lot has changed since the early days of XP and "continuous anything": organizations have moved away from "simple" monolithic applications and embraced distributed and service-oriented architectures. These services depend

on each other in often intricate ways, and interdependencies in distributed systems are among the hardest things to test. If continuous deployment is not done carefully and responsibly, one poorly thought commit has the potential to bring production down.

Continuous delivery practices are a necessary baseline, but some more precautions are needed to perform safe continuous deployment to production, which I already hinted at in “Implications” on page 25. Extra safety nets need to be put in place to guarantee the lowest possible risk of regressions, which have emerged in the years since continuous delivery became popular. Quality gates need to be restructured around the novel proximity of the production environment, and the team’s ways of working need to be adjusted to take immediate deployments into account.

Yet, doing continuous deployment responsibly is possible. Yes, even when not all team members are senior. Teams are doing it every day and enjoying great benefits and incredibly fast feedback from it. In this book, you will find out more about what those benefits are and how they do it. You will learn a framework to perform safe incremental releases during everyday development work that is structured exclusively around the challenges of continuous deployment in nontrivial, distributed systems.

Summary

In this chapter, we introduced continuous deployment: the methodology that will be the focus of this book. In continuous deployment, the entirety of the path to production of code is fully automated—even the final decision to deploy to production.

We then recapped the practices that focused on code’s path to production over recent years: starting from XP’s “If it hurts, do it more often” motto, to DevOps tearing down the barrier between development and operations, to continuous integration introducing automation until an artifact build, continuous delivery extending that to high-level testing and deployments, and finally, continuous deployment removing all human factors from the automated pipeline.

We discussed how continuous deployment has profound ramifications for engineers’ workflow, which needs to be adapted to account for continuous production deployments. We talked about how this results in more granular feedback and more responsibility to implement sturdy quality gates. Commits and deployments need to be planned carefully to keep production in a working and performant state.

In the next chapter, we will see what makes this investment worthwhile.

About the Author

Valentina Servile is lead software developer at Thoughtworks based in Bangkok, working with and advising multiple clients on continuous deployment in distributed systems. She has worked in several cross-functional teams dealing with big distributed systems and microservices, continuous delivery practices, and evolutionary architectures in a variety of tech stacks.

As well as writing code, she enjoys mentoring her colleagues and helping Thoughtworks advise their clients on how to improve their software delivery practices in order to release safely and often—and enable their businesses to respond to change.

Colophon

The animal on the cover of *Continuous Deployment* is a northern fulmar (*Fulmarus glacialis*), also known as an Arctic fulmar. It is a seabird native to the subarctic regions of the northern Pacific and Atlantic Oceans. Though superficially similar to gulls, they are in fact more closely related to petrels and albatrosses. Northern fulmars vary in color from mostly white with gray wings to uniformly gray. They have hooked beaks with two nasal tubes along the top edge.

Northern fulmars have a wingspan of up to 44 inches and can reach 18 inches in length. They are long-lived birds, regularly reaching up to 30 years in the wild, and they typically do not begin breeding until they are 8 to 10 years old. Northern fulmars are opportunistic feeders, eating a variety of marine life as well as carrion and refuse, and they can dive up to 10 feet underwater to catch fish. They have been known to travel up to 600 miles round trip to find food for their young. As a defense against intruders or predators, northern fulmars can spit a foul-smelling oil from their stomachs for several yards.

The northern fulmar is considered a species of least concern. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on an antique line engraving from *Animate Creation*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

O'REILLY®

Learn from experts.
Become one yourself.

Books | Live online courses
Instant answers | Virtual events
Videos | Interactive learning

Get started at oreilly.com.

©2023 O'Reilly Media, Inc. O'Reilly is a registered trademark of O'Reilly Media, Inc. 175 7x9.1975