Mastering Salesforce DevOps

A Practical Guide to Building Trust While Delivering Innovation

—

Andrew Davis

Foreword by Wade Wegner

Apress®

Mastering Salesforce DevOps

A Practical Guide to Building Trust While Delivering Innovation

Andrew Davis
Foreword by Wade Wegner

Apress®

Mastering Salesforce DevOps: A Practical Guide to Building Trust While Delivering Innovation

Andrew Davis
San Diego, CA, USA

ISBN-13 (pbk): 978-1-4842-5472-1 ISBN-13 (electronic): 978-1-4842-5473-8
https://doi.org/10.1007/978-1-4842-5473-8

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Susan McDermott
Development Editor: Laura Berendson
Coordinating Editor: Rita Fernando

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484254721. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Advance Praise for *Mastering Salesforce DevOps*

"DevOps is the next frontier in managing Salesforce orgs for both developers and admins. This book has both the scope and depth to help any organization adapt modern software engineering and management methodologies to Salesforce. Or put another way—if you're a Salesforce developer or admin, you need to read this book now."

—Dan Appleman, Salesforce MVP; author of *Advanced Apex Programming in Salesforce*

"I was amazed at the completeness of this book. The depth and breadth presented here is not available in any single resource that I know of, not even Trailhead. It is not just the Salesforce specific portions either. The DevOps chapters provide a jumpstart for anyone who wants to understand what DevOps is and why it is critically important in today's world."

—David Brooks, VP Products, Copado Solutions; original product owner of AppExchange; one of the three founding PMs of the Force.com Platform

"Andrew Davis has written the essential reference for the next 5–10 years of Salesforce evolution. Salesforce has become a true cloud development platform, but its ecosystem of enterprise-class tools and techniques is just starting to catch up. All prospective and existing Salesforce developers should read this book if they aspire to be not just coders, but professional software engineers."

—Glenn Weinstein, Co-founder of Appirio

"This book really gets to the heart of how to properly equip a team with the tools and process to go faster with the Salesforce platform."

—Sanjay Gidwani, SVP, Copado

ADVANCE PRAISE FOR MASTERING SALESFORCE DEVOPS

"In the last few years it's become clear that for companies to innovate and succeed, IT must have speed and agility like never before. Within Appirio it was Andrew who really caught the wind of the DevOps movement and pushed the entire company in that direction. The power of a performance-oriented culture is that the best ideas can come from anywhere in the company, and it was exciting to watch that unfold as we moved to promote DevOps. I'm delighted that Andrew's insights and passion for this topic are now being shared with the broader world. Those seeking to deliver maximum value from their teams and from Salesforce would be wise to read this book carefully."

—Chris Barbin, Venture Partner, GGV Capital;
Former CEO and Co-founder, Appirio

"Andrew gained experience deploying and developing SaaS applications for enterprise customers while navigating the new world of DevOps in the cloud, and put that into words that can reach readers at any technical level. This book truly delivers a thorough and practical guide to establishing DevOps for those rooted in the Salesforce platform. If you've been hesitant about implementing DevOps, or if you have tried and failed, this is the book for you."

—Katie M. Brown, Director, Methodology at Okta,
Delivery Excellence in the Cloud

"If you are using Salesforce and want to maximize your efficiency to make life easier for your teams, Andrew Davis' *Mastering Salesforce DevOps: A Practical Guide to Building Trust While Delivering Innovation* provides a thorough approach for doing that using Salesforce DevOps. In addition to providing a multitude of technical details, Andrew smartly starts with why—why use Salesforce and why use DevOps specifically for Salesforce—and honestly addresses issues people have had with Salesforce before diving into all the details to help you succeed. Andrew is the acknowledged expert on DevOps for Salesforce and shares his insights and secrets in this book to make your life as a developer, administrator, or user better."

—Dean Hering, Adjunct Associate Professor, Master of Engineering
Management Program, Duke University

"With the rapid pace of development in technology, organizations need to find ways to deliver solutions more efficiently, while retaining high quality. DevOps is an approach to software delivery that ensures both speed and quality, which is important as organizations deploy increasingly complex Salesforce.com solutions. Andrew has dedicated over a half a decade to bring structure and rigor to DevOps on this platform. This is a must-read for anyone working in the software industry!"

—Matt Henwood, Executive Vice President, Service Delivery, 7Summits

"There's never been a more important and exciting time to talk about Salesforce DevOps than right now. Expectations of getting value from an increasingly complex (and saturated) array of Salesforce clouds are high, competition is fierce, talent is scarce, and teams need to modernize to stay relevant. You want the best engine under the hood to make change happen. If you don't, you're doing it wrong. Anyone who deploys anything to production (which is everyone!) needs to read this book."

—Andres Gluecksmann, Vice President, Enable Services, Traction on Demand

"DevOps is becoming a must-have skillset for every developer in today's world. And it is especially crucial with the rapid pace of innovation on the Salesforce Platform. I think this book will become a go-to reference for Salesforce DevOps specialists seeking to leverage all the capabilities of Salesforce DX. Although this book is designed to teach how you can accelerate your Salesforce development lifecycle using CI/CD, even if you are new to Salesforce, the initial chapters will help you to understand the basics of this platform. I had the opportunity to work with Andrew for a couple of years. The best thing about working with him is how many new things you can learn from him, easily and in a very short amount of time. He's been involved with Salesforce DX since it was first announced and this book gives us access to all his learnings and research."

—Durgesh Dhoot, Platform Specialist, Salesforce

To Ashley

"May I always be a manifestation of others' good fortune"

Table of Contents

About the Author

Andrew Davis is a Salesforce DevOps specialist who's passionate about helping teams deliver innovation, build trust, and improve their performance. He is a senior product manager for Copado, a leading DevOps platform for Salesforce. Before joining Copado, he worked as a developer and architect at Appirio, where he learned the joys and sorrows of release management and led the creation of Appirio DX, a set of tools to enable Salesforce CI/CD.

At different times, he led Appirio's technical governance, DevOps, and certification programs and gained 16 Salesforce certifications. An experienced teacher and public speaker, he is a regular speaker at Salesforce conferences. He lives in San Diego with his amazing wife and very cuddly dog. Follow him at `https://AndrewDavis.io` or on Twitter at AndrewDavis_io.

About the Technical Reviewer

John M. Daniel has been working in the technology sector for over 20 years. During that time, he has worked in a variety of technologies and project roles. Currently, he serves as the Director of Platform Architecture at Rootstock Cloud ERP, a leading cloud-based ERP solution that is native to the Salesforce Platform. He is a Salesforce MVP and holds multiple Salesforce certifications, including Platform Developer I and II and most of the Technical Architect Designer certifications. He is currently working toward becoming a Certified Technical Architect. He loves to spend time with his family, swim and ride his Jeep at the beach, and work on open source projects such as Force-DI, AT4DX, and the DomainBuilder Framework. He co-leads his local Salesforce Developers User Group and can be found on Twitter at @ImJohnMDaniel.

Foreword

I've spent my entire career immersed in the world of software development. From early on, I fell in love with the power and the freedom that comes from being able to create magic with code. I've seen the birth of the Internet and have had the opportunity to share this passion for development with thousands of people around the world. But it didn't take long for me to realize that while writing code can be fun, it's not always the most productive way to get the job done. In fact, I've never felt more convinced about the importance of a low-code platform to enable people from every walk of life to experience that same joy and productivity I felt early in my career, creating apps using clicks and, sometimes, code. This is one of the most important things to remember about low-code: a unique and powerful low-code platform like the Lightning Platform lets you combine the power of both clicks and code to do more than you could do with either one alone.

In 2016, I left Microsoft when I was invited to reimagine the developer experience for Salesforce. I realized that this was a unique opportunity to share my love of low-code with the Salesforce community and to accelerate the pace of innovation on Salesforce by giving developers an entirely new way to build together and deliver continuously.

The DevOps movement has developed throughout our industry over the last 10 years into a rallying point for some of the most revolutionary ideas in business and technology. DevOps has a dual meaning, in that it includes a huge range of technological tools and techniques but also speaks to the importance of bringing disparate groups together. Salesforce was founded on the concept of *Ohana*, or community. Just like Salesforce has grown into one of the most passionate and collaborative communities on the planet, the DevOps movement has also inspired passion and a vision for how working together is integral to helping companies perform at levels never previously imagined.

At Dreamforce 2016, I had the privilege to go on stage to introduce Salesforce DX to the world and to share a vision that unites DevOps with the Lightning Platform for the first time. Shortly after that keynote, an earnest and persistent man started following me around the conference. As I walked to make it to a session, he introduced himself and explained that he'd been working on a similar initiative for his company, had anticipated this announcement, and implored me to let him join the pilot. Recognizing his sincerity,

and that he might not leave me alone unless I relented, I invited Andrew Davis and his company, Appirio, to be the first consulting company to join the pilot for Salesforce DX.

I'm so glad I did because the rest is, as they say, history.

This book reflects the dedication and passion that Andrew brings to this topic. And it gathers in one volume all of the core ideas and values that all of us who worked on Salesforce DX have wanted to share but not had the time to write down. In this book you'll see what brought me to Salesforce back in 2016: the power of DevOps and the world's most powerful low-code platform united together. This union is expressed clearly and eloquently in this book.

We're still at the beginning of this journey, both of Salesforce DX and, in a broader sense, of this magical new world that unites the human mind with technology in ways that are both exciting and awesome. DevOps is about working together—about human beings collaborating, working toward a common vision, and using technology to be efficient even at complex activities like building software. As we take these first steps into a new world where the only certainty is change, and where technology increasingly has the ability to determine our future, it is more important than ever that we work together.

This is a technical book. But it's also a human book. It's a book about how to get things done more easily, so we have more time to do what humans do best: to solve problems creatively. It's an honor to introduce this book because I know what's between the covers is the very best of what technology today has to offer: a practical guide to building together on a platform that invites unlimited creativity.

Wade Wegner, SVP Product Management, Salesforce
Redmond, Washington
August 2019

Acknowledgments

It's been a great delight to get to know both the Salesforce community as well as the DevOps community over the last few years. As a lifelong technologist, I've always delighted in the endless puzzle-solving opportunities it presents. But I've been a human even longer than I've been a technologist, and both the Salesforce and DevOps communities are distinctively *human* communities. The degree of openness, collaboration, compassion, and enthusiasm in these communities is inspiring. And when an entire group is inspired, you find what Émile Durkheim called *collective effervescence*, a sense of joy accompanied by a softening of the boundaries between ourselves and others.

That we can be united in a common activity is one of the deepest miracles of being alive. That's also one of the special joys of being part of an organization: that it provides an opportunity for individuals to unite in a shared endeavor. As Peter Drucker said, "The purpose of an organization is to enable ordinary human beings to do extraordinary things." The cloud has enabled larger communities to collaborate in larger endeavors. And DevOps is enabling better coordination and communication in that process. It's no wonder that the Salesforce and DevOps communities are incubating visions for a better world that go far beyond technological improvement.

This book largely captures what I know about this important topic. By sharing my knowledge, I'm also sharing my ignorance, and I welcome any feedback and corrections you have to offer. Every piece of knowledge in this book has come directly or indirectly from others, principally my colleagues at Appirio. I couldn't have hoped for a better place to learn this discipline, and the people who've contributed to my education are too numerous to list.

There are no words to express my gratitude to **my wife, Ashley**, who has been endlessly patient and supportive throughout this learning process. My Sangha jewel, coach, and best friend, she's endured my endless ramblings on this topic and knows far more about both Salesforce and DevOps than she ever wanted to. **My parents and step-parents** lovingly built the foundations for me to be healthy and free and supported me unfailingly even when my decisions led me far away from them physically and culturally. And the **Kadampa community** provided the ultimate opportunity to learn humility, peace, and the joy of living a meaningful life.

From **Appirio** I want to thank the Appirio DX team: **Saurabh Deep**, **Abhishek Saxena**, **Ashna Malhotra**, **Bryan Leboff**, **Rahul Agrawal**, **Katie Brown**, **Kapil Nainani**, **Sahil Batra**, and **Durgesh Dhoot**. You all believed in this vision and did the real work to make the project a reality. To **Yoni Barkan**, **Roarke Lynch**, **Rebecca Bradley**, **Halie Vining**, **Craige Ruffin**, **Erik Golden**, and **Katie Boehner**—you all are the real deal; I'm sorry we weren't able to work together longer. I'm grateful to my other mentors and teachers at Appirio, especially to **Geoff Escandon** who brought the *State of DevOps Report* to my attention and challenged the early work I was doing saying "I don't know what this is, but it's definitely not DevOps." I hope I'm getting closer.

It was **Glenn Weinstein**, **Chris Barbin**, and **Erik Duffield** at Appirio who championed this project at the highest levels and who fostered a performance-oriented culture in the company from the beginning. My coworkers at **Oath** also deserve special recognition for introducing me to continuous delivery. In particular, **Matt Hauer** dissuaded me from leading the team into branching hell and never let me ignore a broken build. **David Meyer** first challenged me to deliver "CI for the masses," a project I'm still working on. **Matt Henwood** challenged me to "let my creative juices flow" and ran interference for me as this project got off the ground. **Bob Larson** asked for some short-term assistance to set up CI/CD for a big customer; two years and thousands of deployments worth of short-term assistance gave me the confidence to write this book. I'm particularly grateful to my partners in that endeavor, **Alex Spears** (who endured the misery of being mentored by me), **Sreenath Gopal**, and **Raji Matthew**. Special mention is also due to **Lex Williams**, **Randy Wandell**, **Joe Castro**, **Andres Gluecksmann**, **Chris Bruzzi**, **Michael Press**, **Svatka Simpson**, **Neale Wooten**, **Jitendra Kothari**, **Prakash Gyamlani**, **Tommy Noe**, **Tommy Muse**, **James Wasilewski**, **Norman Krishna**, **Josh Davis**, and everyone else who supported our DevOps initiatives in a hundred ways.

It's a unique honor to work with **John M. Daniel** as the technical reviewer for this book. To have this work reviewed by a mind as sharp and experienced as his gives me far greater confidence that I'm not making this stuff up. And I'm grateful for the team at **Apress**, especially **Susan McDermott**, **Rita Fernando**, and **Laura Berendson** for affording this opportunity and for doing the hard work of bringing a book like this into existence.

Finally, to my new colleagues at **Copado**, especially **David Brooks**, **Andrew Leigh**, **Ted Elliott**, and **Federico Larsen**, the fun's just getting started. I look forward to working with you to help thousands of organizations master Salesforce DevOps.

CHAPTER 1

Introduction

This book provides a practical guide to bringing DevOps principles to Salesforce development. For those whose careers are rooted in the Salesforce world, this guide will help you adopt practices that have been proven to make life easier and teams more effective. For those who specialize in DevOps or other technologies and are tasked with adopting or supporting a Salesforce implementation, this book will allow you to translate familiar concepts into the unique language and practices of Salesforce. And for those who are already under way with optimizing your Salesforce development process, I hope that this book provides inspiration to go further, think deeper, and continue to embody excellent practices on an exceptional platform.

Why Salesforce?

Over the past 20 years, Salesforce has become the fastest growing enterprise software platform and a market leader in Sales, Service, Marketing, Integration, and custom application development among other areas.[1] Although there are many reasons why a company might adopt or consider using Salesforce, the overriding value proposition is that Salesforce provides tons of capabilities out of the box, releases new capabilities at an ambitious pace, and allows for virtually endless customization and innovation. Unlike with traditional platforms, there's no need to manage infrastructure for computing, data, network, or security and no need to invest in generic system administration that adds no value for the organization. Add to this the accessible and supportive community and learning culture based around Salesforce's Trailhead, which enables rapid onboarding and skill development, and it's easy to see the attraction of this platform.

[1] https://s1.q4cdn.com/454432842/files/doc_financials/2018/Salesforce-FY18-Annual-Report.pdf

A. Davis, *Mastering Salesforce DevOps*, https://doi.org/10.1007/978-1-4842-5473-8_1

Companies who adopt Salesforce do so because they don't want to manage servers, networking infrastructure, security, and the countless other aspects of application development that don't directly bring business value. When an enterprise company takes an interest in automating a new part of their workflow or storing data, they have no interest in provisioning servers or containers, let alone implementing ongoing monitoring, data backup, failover redundancy, and so forth. What they're interested in is business value. And what Salesforce delivers is the opportunity to create and deploy business value with the minimum necessary overhead. In exchange for a relatively modest subscription price, Salesforce takes care of almost every aspect of the infrastructure needed to run an application. At the same time, they have continued to expand and innovate to make the process of developing functionality easier across a broadening range of applications.

Salesforce began in 1998 with the aspirational goal to create a SaaS customer relationship management (CRM) system that was "as simple as Amazon."[2] Since then, they've become the fastest growing enterprise software company ever to reach $10 billion. And they lead the market for CRM, service automation, marketing automation, and integration. They've been recognized year after year by *Forbes* as the world's most innovative company and won their Innovator of the Decade award. *Fortune* magazine also ranked them the #1 best place to work.[3]

Salesforce's mantra throughout the 2000s was "Clicks not Code," and their logo was the "No Software" image which became embodied as SaaSy, Salesforce's first mascot. You could be a "declarative" developer using "clicks" alone and build a data model, business process automation, and user interface using drag-and-drop interfaces. This opened the door to an entire new job specialization—Salesforce Admins and "App Builders," citizen developers who were empowered to directly customize the platform to meet their company's needs.

As Salesforce grew, they recognized the need to enable custom coding on their platform and launched Apex triggers, followed by Apex classes, and Visualforce in 2008. Admins continued to configure rich business capabilities, while low-code developers and professional developers were empowered to create customizations beyond the scope of what could be built declaratively. Initially, the code developed on Salesforce was rudimentary: a few hundred lines here and there to accommodate unusual

[2]Marc Benioff, *Behind the Cloud* (Wiley-Blackwell, 2009).

[3]https://s1.q4cdn.com/454432842/files/doc_financials/2018/Salesforce-FY18-Annual-Report.pdf

calculations and data processing. But gradually, companies began to accumulate tens of thousands of lines of custom code, especially as legacy applications were migrated from other systems. Needless to say, the "No Software" motto no longer strictly applies.

In the decade since the launch of Apex, 150,000 companies have adopted Salesforce and moved a staggering number of legacy systems onto Salesforce. Salesforce now processes billions of transactions per day. Many of these transactions are part of the core CRM and customer service applications, but a huge number relate to custom applications, making Salesforce one of the largest Platform as a Service (PaaS) providers.

What Is Salesforce DX?

Salesforce's ease of customization has led to an unusual challenge. Whereas developers in most languages have been employing methods such as continuous delivery for a decade or two, most Salesforce customers have been slow to adopt such methods. There are two reasons for this: the Salesforce platform itself and the developers managing it.

First of all, the Salesforce platform behaves very differently from a standard server or custom application. You can't build a local instance of Salesforce on a developer's laptop, for example. Instead, Salesforce runs centrally "in the cloud," and you deploy functionality by updating a specific Salesforce instance. There are limits to what types of changes can be deployed, and tracking changes has been challenging, meaning that developers often needed to log in to a target org and make customizations manually to complete a deployment.

The other reason the Salesforce world has been slow to adopt practices such as continuous delivery is that Salesforce development is a specialized skill requiring declarative/admin skills, with programmatic/coding skills being a nice-to-have addition in many cases. Because most Salesforce developers don't do custom development on other platforms, this led to a lack of cross-pollination between ideas that were common in the Java, JavaScript, .NET, and Ruby worlds. Even when traditional developers were transplanted into a Salesforce project and reskilled to work on Apex or Lightning, they found that their customary build, test, and deploy tools were largely useless in the new Salesforce world. Instead they were provided with "change sets," an Ant Migration Tool, and very few examples of how one might build a comprehensive CI/CD pipeline for this platform. The brave few ventured into configuring complex Ant "targets" using XML and triggering them using Jenkins. But by and large, teams cobbled together a mix of manual

steps and light automation, tackling deployments one at a time, and as infrequently as they could get away with.

The challenge of managing Salesforce deployments was one factor that led Salesforce to earn a very dubious accolade. In the 2017 Stack Overflow international developer survey, Salesforce tied with SharePoint as the "most dreaded platform to develop on."[4] Ouch. Needless to say, that is not one of the accomplishments Salesforce is most proud of.

When companies were using Salesforce solely for sales force automation, it could be a "shadow IT" product managed by the Sales Operations teams. But as its capabilities and adoption expanded beyond CRM, Salesforce has moved to being a core part of companies' IT infrastructure. An increasing number of CIOs and IT Directors recognized the centrality of Salesforce to their businesses and the growing complexity of their Salesforce implementations. Salesforce was increasingly fielding challenging questions in terms of how companies could adopt DevOps best practices, such as maintaining and deploying all configuration from source control.

These demands eventually led Salesforce to establish an internal tiger team known as "Salesforce DX," dedicated to improving the Salesforce Developer eXperience. The team was assembled in 2016, with several seasoned groups working in parallel on improving different aspects of the developer experience. At Dreamforce 2017, Salesforce DX went live.[5]

Prior to Salesforce DX, the only way to develop on Salesforce was org-based development, where a Salesforce org itself is the source of truth. Even if teams managed to use version control and automated deployments to update their testing and production orgs, version control necessarily lagged behind the development org. This meant that changes to the development org were often never tracked in version control, and that it was entirely possible to make changes in the development org that were difficult or impossible to track and deploy systematically. Since the org was the source of truth, anyone wanting to see the latest, integrated version of a component had to check the org, since version control could always be lagging behind.

Org-based development meant that even the most sophisticated development teams were trapped in a conundrum: they could provide each developer with their own separate org, or they could have all developers work in a single org. Separate orgs meant

[4] 2017 Stack Overflow Developer Survey, https://insights.stackoverflow.com/survey/2017#technology-most-loved-dreaded-and-wanted-platforms

[5] www.salesforce.com/video/317311/

it was easy for developers to isolate their changes from the work of others, but orgs quickly got out of sync. Sharing a single org provides continuous integration (developers' work is always integrated in a single org), but it is extremely challenging to track and deploy one's own changes in isolation from the rest of the team. Vendors such as Flosum and AutoRABIT offer sophisticated synchronization tools, but the whole system is fundamentally flawed.

The most notable innovation to launch with Salesforce DX is a new type of Salesforce org called a scratch org. For the first time, scratch orgs allow a Salesforce environment to be created entirely from version control. This means that for the first time, version control can be the source of truth, rather than always lagging behind the state of development orgs. Developers can create private development environments at will, and those environments can be quickly updated from version control as other members of the team integrate their own changes.

Salesforce DX thus finally allows for source-driven development. Source-driven development in turn opens the door to adopting most of the other technological and cultural processes known broadly as DevOps. And DevOps has been shown to be a massive contributor to innovation, effectiveness, and corporate success.

What Is DevOps?

This book represents a convergence of two major movements within the IT world: the movement to Software/Platform as a Service (SaaS/PaaS) and the DevOps movement. The goal of SaaS and PaaS is to provide a scalable, global, high-performing foundation for companies to invest in their core competencies rather than having to maintain their own infrastructure. In addition, centralized SaaS applications provide powerful capabilities without introducing software upgrades, version incompatibilities, and the other headaches associated with maintaining your own software.

Salesforce provides SaaS solutions for customer relationship management (CRM), sales force automation (SFA), customer service, and more, while also offering a PaaS platform for deploying custom code (Heroku and the Lightning Platform). Salesforce DX now provides the raw capabilities to enable a DevOps workflow on Salesforce. The goal of this book is to explain how you can build a powerful and comprehensive DevOps workflow for Salesforce. The power of this can't be overstated: you can finally deploy the world's most innovative platform using the world's most effective and efficient techniques.

Within the literature and tooling providers for DevOps, there is a tremendous focus on enabling Infrastructure as a Service (IaaS) as an evolution away from managing your own servers. "DevOps solution providers" include a massive array of IaaS providers (such as AWS, Google Cloud, and Azure), as well as tools for managing software packages, database configurations, application monitoring, and so on. Almost every "DevOps tool" provider is addressing a need in the IaaS space so that organizations with legacy applications and infrastructure can begin to manage those in an efficient and orderly way.

It's interesting to note that almost all of these "DevOps tools" are helping to solve problems that simply do not exist for Salesforce users. Simply moving to Salesforce solves most every problem with infrastructure, security, monitoring, databases, performance, UI, and more. One reason Salesforce users have been so slow to adopt DevOps practices is that most of them are simply not needed. Sort of.

The reality is that functionality developed on Salesforce simply exists at a higher level: application configuration instead of infrastructure provisioning. You basically never have to worry about Oracle database patches or that Salesforce will lose your data; but that doesn't solve the problem of how to ensure that your database fields are consistent across all of your Salesforce instances. The exact same principles championed across the DevOps community are entirely applicable in the Salesforce world, they simply need to be adjusted to the particulars of the Salesforce platform.

DevOps can be understood as bringing together the practices of continuous delivery (and all that entails, such as version control) with principles of lean software engineering and an inclusive focus spanning from development to operations. DevOps builds on principles that have evolved over decades in manufacturing and software engineering and brings them together under a surprisingly catchy moniker.

Version control, continuous integration, continuous delivery, DevOps—what's that all mean? Version control is a mechanism to track and merge changes smoothly. Continuous integration means that teams should develop their work on a common master branch, minimize branching, and run automated tests after every change to the master branch. Continuous delivery means that all the configuration needed to recreate your application is stored and deployed from version control in an automated way. And DevOps means that both developers (who create new functionality) and operators/admins (who maintain production systems) should work together to optimize the flow of valuable work to end users, build software with monitoring in mind, and use the same mechanisms (such as version control) to both maintain and improve applications.

DevOps has been summarized by Jonathan Smart as "Better value, faster, safer, happier." "Better" implies continuous improvement, always striving to improve the quality of not only our work but also our processes. "Value" means using agile development and design thinking to address and adapt to the needs of end users. "Faster" means using techniques such as continuous delivery to release more quickly. "Safer" means integrating automated quality and security scanning into the development workflow. And "happier" refers to continuously improving both the development experience and the customer experience of users.

It's worth noting that the "DX" in Salesforce DX stands for "Developer Experience." The implication is that Salesforce is investing in providing a better experience for those developing on its platform, not just for its end users. Although the Admin/App Builder experience for Salesforce has always been very good, professional developers have historically been an afterthought. As a developer who moved onto the Salesforce platform, I found it confusing and frustrating that version control was not a common practice, and that Salesforce lacked sophisticated tools such as autoformatting and ESLint that make it easier to write good code in JavaScript.

In *The Phoenix Project*[6] and *The DevOps Handbook,*[7] Gene Kim popularized the idea that there are "three ways of work" or three movements that summarize DevOps. They are continuous delivery, the "left-to-right" movement of features and fixes from development into production; continuous feedback, the "right-to-left" movement of feedback from testers and production to developers; and continuous improvement, the ambition to always improve the "system of work" that is used to deliver the work itself.

Traditionally, software development was depicted as a linear process similar to an assembly line. But the DevOps community often uses variations on a circle or infinity loop as shown in Figure 1-1 to indicate that software development must be iterative and ongoing. Each stage in the development lifecycle flows into the next in an ongoing pattern. By spanning the entire lifecycle from planning and development to operation in production, DevOps promotes collaboration across teams to maximize the entire value chain or flow of valuable work to end users.

[6]Gene Kim, Kevin Behr, and George Spafford, *The Phoenix Project* (IT Revolution Press, 2013).

[7]Gene Kim, Patrick Debois, John Willis, and Jez Humble, *The DevOps Handbook* (IT Revolution Press, 2016).

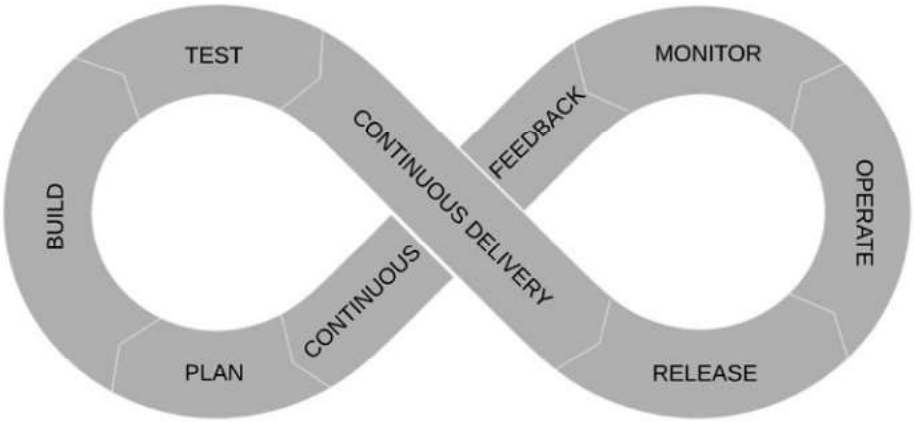

Figure 1-1. *A DevOps infinity loop*

The Research on DevOps

Since 2012, Puppet Labs has produced an annual survey and report entitled the "State of DevOps Report." That report has become increasingly methodical and influential, especially as Puppet Labs was joined by DevOps Research and Assessment (DORA) group. In 2018, the cumulative research from this report was published as the book *Accelerate* by Nicole Forsgren, Gene Kim, and Jez Humble (co-leaders of DORA).[8] In that book (as well as in the annual "State of DevOps" Reports), they provide scientific verification of long-held convictions that adopting DevOps practices predicts software delivery performance, positive organizational culture, and organizational performance (both financial and nonfinancial).

Accelerate lays out the research that use of continuous delivery (version control, CI/CD, etc.) leads to increased software delivery performance, which in turn improves corporate performance. High-performing teams are able to spend 66% more of their time doing new (constructive) work and 38% less time on issue resolution and unplanned work compared to low-performing teams. In addition, teams with high Software Development Performance (SDP) exceeded those with low SDP with

- 46 times more frequent code deployments
- 2,555 times shorter lead time from commit to deploy
- 7 times lower change failure rate (1/7 as likely for a change to fail)
- 2,604 times faster mean time to recover from downtime

[8]IT Revolution Press, 2018.

Software Delivery Performance also predicts corporate performance. High software delivery performers are twice as likely to exceed their commercial goals (profitability, market share, and productivity) and their noncommercial goals (quantity of goods and services, operating efficiency, customer satisfaction, quality of products or services, and achieving organization or mission goals).

Another long-standing belief in the DevOps community is that positive organizational culture leads to great results for the company. Salesforce themself has emphasized such a positive organizational culture from its outset, and many organizations in the Salesforce ecosystem such as Appirio also embody principles such as openness and philanthropy. The *State of DevOps Report* confirms the importance of culture for organizational performance. That same survey confirms that the use of continuous delivery itself predicts a positive organizational culture, high job satisfaction, high delivery performance, and employee engagement. The use of continuous delivery (version control, CI/CD, etc.) also leads to less rework, less deployment pain, and thus less burnout.

About This Book

This book is divided into four parts. **Part 1: Foundations**, provides an introduction to Salesforce and to DevOps concepts. Those new to Salesforce will find Chapter 2: Salesforce to be a helpful overview of the Salesforce platform, including core concepts and challenges that might not be obvious at first.

Those who are new to DevOps will find Chapter 3: DevOps to be a concise summary of the latest best practices and research on this topic. There is nothing in that Chapter that will be new or groundbreaking for those who are very familiar with DevOps, although it may provide a helpful recap and summary. The reason for providing a comprehensive overview of DevOps is to provide a single reference for people in the Salesforce world who may be unfamiliar with these practices. Books such as *Continuous Delivery* by Jez Humble and David Farley[9] will allow readers to go much deeper into these concepts.

The last three parts of the book cover the Salesforce development lifecycle, all summarized into three stages: development, delivery, and operations. These three stages are common to all software development, and are in fact common to manufacturing

[9]Jez Humble and David Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation* (Addison-Wesley Professional, 2011).

and product development in general. Things are made, then they're shipped, and finally they're used. Along the way, they're generally *bought* as well, which implies that what is being created has value.

The only reason things are made is in the hope that they will bring greater benefit than they cost to make or buy. The difference between the benefit something brings and the cost of making it is known as value added. Lean software development, and lean practices in general, talk extensively about value creation, value streams, and value-added steps, in contrast to what is known as waste. The goal of DevOps is to reduce waste, and to enable teams to deliver value as effectively as possible.

Thus when we talk about development, delivery, and operations, we are primarily talking about the stages in which value is created (in the form of new software functionality), delivered (including testing and deployment), and realized or experienced by end users. **Part 2: Salesforce Dev** provides a concise overview of how to maximize the ease of developing on Salesforce and the quality of the work. Of particular interest is how to develop so that the subsequent testing, deployments, and operations can be as successful as possible. The very definition of DevOps implies that developers keep operation in mind. That means that things are developed so they can be delivered as quickly as possible and operated successfully, and that defects can be discovered and features improved in the most efficient way.

Part 3: Innovation Delivery covers the stages in which Salesforce DX really shines. Software deployments in general are painful; and Salesforce deployments have been notoriously painful. The pain of deployments drives people to do them as infrequently as possible. This means that in addition to being delayed to once a week or once a month (or less), deployments are done in large batches. A fundamental concept in lean software development is to develop and deploy in small batches. This enables the highest value features or fixes to be deployed as soon as possible, reduces the risk and impact of each deployment, and opens the door to fast feedback.

Deployments aren't just about moving functionality into production; moving functionality into testing environments also requires deployments. So delays in deployment imply delays in testing. Delaying the start of testing doesn't always mean that go-live can be delayed, so the QA and testing process in Salesforce projects is often squeezed. The high pressure to test quickly to avoid further delays in feature delivery has meant that the entire testing ecosystem in Salesforce is generally underdeveloped, with an overemphasis on repetitive manual testing. Part 3: Innovation Delivery provides an overview of how to deploy, test, and release Salesforce features, including both older technologies and the newer capabilities such as unlocked packages that Salesforce DX provides.

Part 4: Salesforce Ops (Administration) is on operations, or how Salesforce capabilities are used in production. As many people have commented, software spends most of its life in production, so it makes sense to consider that phase carefully, including how to measure and detect issues as quickly as possible. On many platforms, operations is equivalent to administration. But in the Salesforce world, the term "Admin" is overloaded to mean both "keeping the lights on" and building new capabilities declaratively. Salesforce's "Awesome Admins" are thus responsible for the majority of development on the platform, in the form of adding objects and fields to the database, modifying business logic declaratively, and changing the UI through layouts. The tension between developers and operations in Salesforce is ironically the opposite of how it is on most platforms. On most platforms, it is the admins who are screaming to keep the system stable and running, while developers are clamoring to release new capabilities. In Salesforce, however, it is admins who are more likely to make changes directly in production at the behest of users and to be reluctant to follow the disciplined deployment processes that developers are forced to adhere to. Locking admins out of production (at least in the sense of preventing them from changing the database, business logic, or UI) is actually a requirement for realizing the goals of DevOps. But for this to be palatable, there has to be a smooth and admin-friendly process in place whereby small declarative changes can easily be made, committed, and deployed to production. Thus the final section speaks to how admins can participate in the DevOps process.

For some companies who have adopted continuous delivery for Salesforce, the first skill new Salesforce admins are taught is how to use Git, so that they can make their changes in a development environment, but track and deploy them by themselves. And for those who are skeptical whether their admins will ever adopt Git, we'll discuss the variety of options available to help Admins keep their orgs stable while still churning out business value at the speed of clicks.

Background

This book arose out of a 4-year focus on the Salesforce development lifecycle, especially Salesforce DX, while working at Appirio. Appirio is a consulting company focused exclusively on cloud technologies, primarily Salesforce. From the beginning, the company has emphasized effective development processes and worked hard to bring customers the most value in the shortest amount of time. I joined Appirio in 2014 as a technical consultant and very soon began evangelizing the use of version control and

continuous integration for Salesforce. At the time those practices were quite rare on Salesforce projects. But my colleagues and mentors at Appirio regularly encouraged me that this was indeed a critical need across most projects. My focus became how to adapt these practices to the peculiarities of Salesforce and to help others do the same.

In 2017, after 2 years of working on side projects related to code quality and development automation, Appirio agreed to establish an internal team known as Appirio DX to focus on tooling, training, and evangelism to bring DevOps principles to all of our projects. In 2018 we released Appirio DX as a commercial tool that could be used by anyone to help ease the Salesforce development process. In 2019 I joined Copado, one of the leading DevOps tool providers for Salesforce, to work on the next generation of their platform.

I've been inspired by Appirio and Copado's commitment to empowering developers and admins, and am grateful for their providing an environment in which I could learn so much and be free to chase the wild but important idea that DevOps is critical to delivering innovation while building and maintaining trust.

PART I

Foundations

Before getting into the details of how to implement DevOps on the Salesforce platform, it's important to establish some foundations. What is Salesforce, and how is it different from other kinds of application or platform? And what do we mean when we talk about DevOps?

CHAPTER 2

Salesforce

It's important to clarify what we mean by "Salesforce" in this book, since the company has grown by acquisition, and the techniques shared in this book do not apply to all of the products that now fall under the Salesforce brand.

What makes Salesforce different? How is DevOps done in the Salesforce world? What is Salesforce DX? And how does it facilitate DevOps on Salesforce?

"Salesforce" vs. Salesforce

Salesforce is a vast and growing company with a vast and growing suite of products. It's a challenge even to keep up with the names of the various products that Salesforce releases or attains through acquisition.

The focus of this book is on what's known internally at Salesforce as "Salesforce Core." Salesforce Core consists of Sales Cloud (the core CRM application), as well as Service Cloud (for customer support), Community Cloud (to create customer-facing web applications), and the Lightning Platform (previously known as `Force.com`). The first three of these constitute the SaaS part of Salesforce, while the Lightning Platform constitutes the PaaS component. All of these components of Salesforce Core work together seamlessly.

In the architecture of Salesforce, "Salesforce Core" is a massive, multi-gigabyte JAR file that is deployed across data centers and "Pods" around the world to allow secure multitenant access to customers via the Internet.

A. Davis, *Mastering Salesforce DevOps*, https://doi.org/10.1007/978-1-4842-5473-8_2

The "markitecture" of Salesforce (how Salesforce is presented externally by their marketing department) depicts a vast and cohesive range of other products such as Tableau, Heroku, Marketing Cloud, and Commerce Cloud, but these are all truly separate products, generally the result of acquisitions. Sharing data or functionality between these products and Salesforce Core requires some type of integration.

Although Salesforce is working hard to integrate these products in a way that is transparent to customers, these other products are developed and deployed in fundamentally different ways. For example, the methods of developing and deploying customizations to Marketing Cloud or Commerce Cloud are entirely different from the way customizations are developed and deployed to Salesforce Core.

Therefore, throughout this book, whenever we refer to "Salesforce," we are referring either to the company itself or to this Core platform, and not to any of these other products.

How Is Salesforce Different?

As mentioned earlier, Salesforce provides both SaaS (Software as a Service) and a PaaS (Platform as a Service). By contrast, much of the focus elsewhere in the DevOps world is in moving from on-premise infrastructure to using IaaS (Infrastructure as a Service). Figure 2-1 shows an illustration of the differences between these four modes and how they represent progressive simplifications of what companies themselves have to manage.

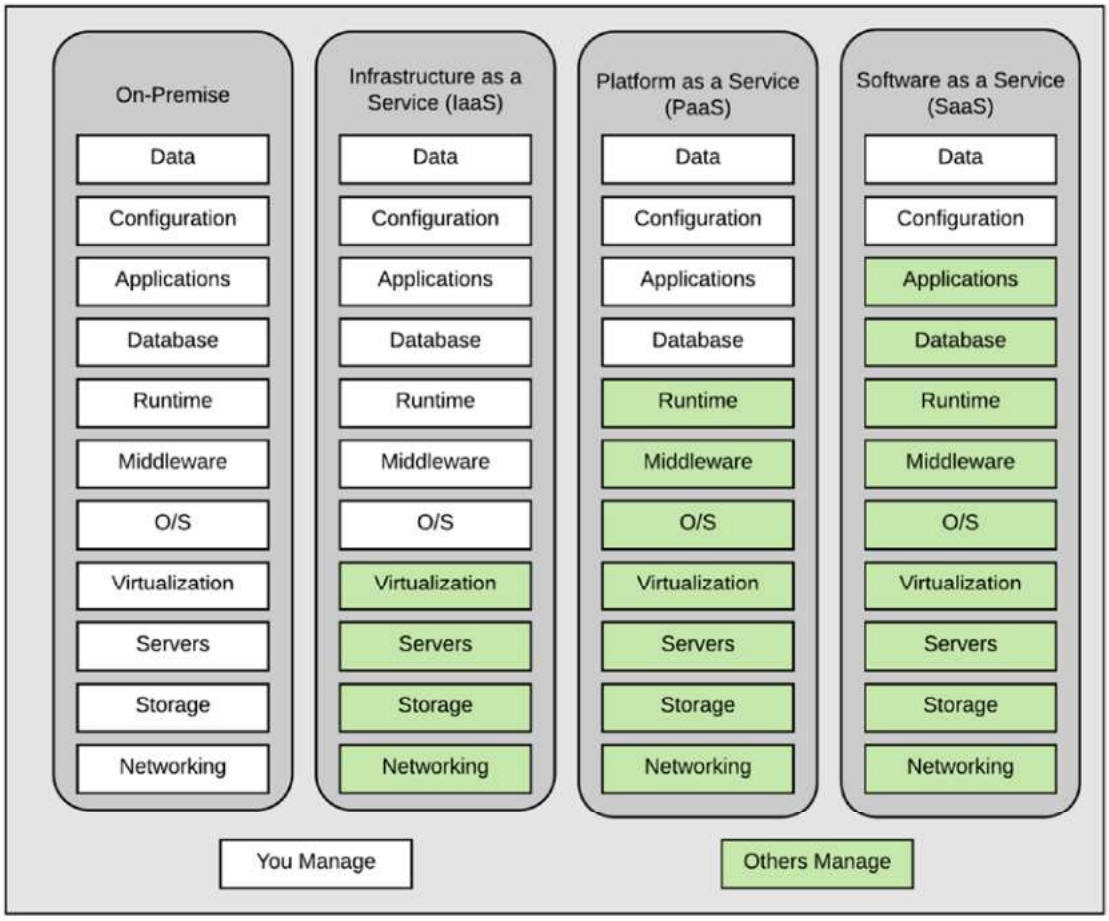

Figure 2-1. *On-premise systems require you to manage all of the resources yourself. IaaS, PaaS, and SaaS delegate progressively*

Managing the Salesforce development lifecycle requires a unique skillset and approach compared to most other platforms. You can say that the DevOps movement was jointly realized by both developers (working in traditional languages such as Java) and system admins or operators (working on traditional infrastructure such as servers and databases). There are thus a vast range of well-established tools and techniques in the DevOps world for developing and deploying code and for managing and updating infrastructure. Unfortunately, almost none of them can directly be used for Salesforce.

To illustrate this, let's look at how teams would manage infrastructure using AWS or deploy an application onto Heroku (a PaaS product also owned by Salesforce). Every aspect of AWS infrastructure can be represented using JSON configuration. JSON can be used to define which AWS services are used, which data centers they are running in,

and how they're configured. The AWS CLI (command-line interface) can be used in a continuous integration tool such as GitLab CI to automatically deploy updates to AWS infrastructure every time the JSON configuration changes.

Similarly, Heroku provides a platform to deploy custom application code (in Java, PHP, Ruby, etc.) and to specify which services (such as Postgres databases) are needed to support that application. The Heroku CLI can be used to automatically update the application whenever the codebase changes. Heroku also provides a tool called Pipelines that allows you to visualize and manage the entire development lifecycle without requiring a third-party CI tool.

AWS's "infrastructure as code" approach is a delight for sysadmins, since it allows them to track changes and automate updates. Similarly, Heroku removes the vast majority of the setup and dependencies that developers would need to deploy their applications. Heroku provides a true Platform as a Service that is ready to receive custom-coded applications built in a traditional way: from the ground up.

Although we can say that Salesforce's Lightning Platform is a PaaS, it actually works extremely different from true PaaS systems such as Heroku. The Lightning Platform allows you to write custom server-side or client-side code, but that code can only run on Salesforce. Although Salesforce allows you to define custom database tables and fields, that schema cannot be loaded into any database other than Salesforce's. These customizations are effectively just changes to the configuration of one Salesforce instance. It's therefore more accurate to say that every customization you can make to Salesforce is basically a Salesforce configuration change.

Salesforce is actually just a big application that happens to allow for infinite customization. But this means that the tools used for managing other IaaS and PaaS products cannot be used to customize Salesforce. Fortunately, however, the release of Salesforce DX means that the *techniques and principles* used with other technologies can now be ported over to Salesforce. That is the focus of this book.

DevOps in the Salesforce World

Moving from on-premise CRM software to Salesforce removes the need for servers and manual software upgrades. Consolidating customer support and online community management onto the CRM platform removes the need to integrate sales, support, and community applications. Migrating legacy applications onto the Salesforce platform allows those applications to share data and processes with the rest of the business.

Because moving to the cloud dramatically simplifies many of the challenges in delivering IT functionality, many Salesforce customers have been able to innovate quickly without having DevOps practices in place. But even without the hassle of managing servers and building software from scratch, the inevitable growth of complexity eventually causes companies to struggle with issues like orgs getting out of sync and delays in deploying functionality.

Salesforce solves so many IT headaches, it's almost easy to overlook the fact that it's created some new ones.

Salesforce has unleashed the ability for companies to focus on their core competencies, instead of struggling to provide basic IT services. Salesforce admins and developers are empowered to spend their time directly creating business value. But with multiple Salesforce admins and developers working at companies year after year, more and more Salesforce customers are finding themselves drowning in all of that "business value."

If you have tens of thousands of components, each providing business value, but no systematic way of tracking, managing, or deploying them, you now have a new form of business pain and chaos. Far too much of a good thing.

What Is Salesforce DX?

Salesforce DX is an initiative begun by Salesforce in 2016 and that launched publicly at Dreamforce 2017. "DX" here stands for Developer eXperience. The goal of the initiative is to reenvision the developer experience on Salesforce with a focus on how to empower the Salesforce development community with the tools and processes needed to develop safely and effectively.

Salesforce DX is a very broad initiative, including several teams at Salesforce focused on environment management, custom coding, developer tooling, APIs, and more. Although the Salesforce DX teams tackle very diverse needs, their main focus (at least initially) is to improve the developer tooling and development lifecycle.

The tools and capabilities included in Salesforce DX are made available for free to all Salesforce customers. In that sense, Salesforce DX is an important complement to Trailhead, Salesforce's free, self-paced, gamified, interactive learning platform. Trailhead is also a major investment for the corporation. Together these are intended to support the growth of skilled Salesforce professionals and to make it easier for those workers to build and innovate on the platform. These multiyear strategic initiatives help ensure that talent shortages and worker inefficiency will not be the limiting factor on the company's aggressive 30% annual growth.

What Are the Elements of Salesforce DX?

Salesforce DX is a new way of developing and collaborating on Salesforce. At its heart are two main concepts:

- Version control is the source of truth for each project. Development is done principally in scratch orgs (temporary Salesforce environments created from version control) as opposed to Developer sandboxes. Scratch orgs are discussed more in Chapter 6: Environment Management.
- Code and metadata should be subdivided into packages representing discrete functionality. This reduces complexity and allows packages to be developed and deployed independently of one another. unlocked packages are discussed more in Chapter 9: Deploying. In addition, Salesforce DX includes new developer tools: the Salesforce CLI and a set of extensions for the Visual Studio Code IDE. The Salesforce CLI replaces an earlier set of Ant-based tools called the Ant Migration Tool, while the VS Code extensions replace the original Force.com IDE built on Eclipse. Salesforce's choice to build on Visual Studio Code was prescient, since that free tool has come to dominate the code editor marketplace.

All of these capabilities and tools are built on top of Salesforce's existing APIs such as the Metadata API, and so continuing to improve those APIs has also been the responsibility of the DX team. A persistent challenge for teams hoping to automate their development lifecycle has been that some configuration settings can't be deployed automatically. Meanwhile, the platform continues to expand its capabilities. Many Einstein and Community capabilities were not initially supported by the APIs. A major victory for the DX team was enforcing automated processes behind the scenes to ensure that all new features on the platform are API accessible when they are released.

In addition, the DX teams have done a significant amount of developer outreach and evangelism to help customers understand how to take advantage of these capabilities. It's my hope that this book can also be of benefit in this effort.

The Dev Hub

Some aspects of Salesforce DX are installed on a developer's local machine or CI runner, while other aspects are capabilities built on top of Salesforce itself. The capabilities built into Salesforce include the ability to create new, ephemeral Salesforce orgs called scratch orgs and also to create and publish versions of unlocked packages. To make use of these, a development team needs to designate a production Salesforce org to be a Dev Hub and enable the "Dev Hub" setting in that org.

Enabling Dev Hub has no risk or side effects to the org. And since scratch orgs created from that org carry no data or metadata, giving developers access to a Dev Hub does not constitute a security risk. There is even a free license type called "Free Limited Access License" that can be enabled in a production org to allow developers or consultants to use a Dev Hub even if they wouldn't otherwise have a user license in that org. This license type does not expose any data or metadata in the production org; it just gives access to use the Dev Hub functionality.

Each developer also needs appropriate permissions on that Dev Hub (see "Permissions Needed to Use the Dev Hub" in Chapter 6: Environment Management). For training purposes, Developer Edition orgs, including those created in Trailhead, can be used as Dev Hubs, although their limits are too restrictive for production use.

Scratch Orgs

As mentioned, scratch orgs are temporary, disposable environments similar to virtual machines. They are populated with code and metadata stored in version control and can be used to support development, testing, and continuous integration. These orgs live for only 7 days by default, although they can be configured to live up to 30 days. Scratch orgs are used for source-driven development, while sandboxes remain useful as long-running test environments. Packages and metadata are developed in scratch orgs and can be deployed to sandboxes and production using CI/CD.

Second-Generation Packaging

One of the most important characteristics of our bodies is the existence of many internal organs that each perform specialized functions while also working together as a whole. Well-organized software follows a similar pattern, known as modular architecture. The division of software into independent modules is a long-standing best practice that makes software easier to understand, maintain, and update.

Modular architecture in software exists at many levels, such as methods, classes, files, and packages. Each of these represents a layer of abstraction that brings similar benefits. The general idea is that the details of what is *inside* the module should not concern other parts of the system *outside* the module. What is of concern is the *interface* between that module and the rest of the system, for example, the input parameters and return values for a method. This allows each module a degree of independence, to be internally changed and refactored, as long as they don't change their interface with the rest of the system. Similarly, when viewed from outside the module, all that matters is its interface. By hiding the underlying details, the whole system becomes easier to understand and work with.

Modules that comprise collections of many files that all perform related functions are called packages. Most if not all high-level software languages support different types of packages such as JavaScript modules, Ruby Gems, or .NET NuGet packages. This type of packaging is enormously helpful for developers since it allows them to build on pre-packaged solutions such as any of the 800,000 JavaScript modules on NPM[1] rather than attempting to recreate such solutions themselves.

Salesforce has long supported the creation and installation of managed and unmanaged packages on its platform. Managed packages are typically used by ISVs to create commercial applications, since they hide their internal IP and prevent most functionality from being modified. Unmanaged packages are often used as a mechanism to share or install groups of related functionality. The AppExchange has been a market-leading business "app store" since its inception,[2] and most of the "apps" available there are in fact managed or unmanaged packages that can be installed in a Salesforce org.

The challenge is that although unmanaged packages can be used to install related functionality, the metadata within that package does not remain part of that package. The package exists like a cardboard shipping container that does not perform any useful function once it has been delivered and unpacked.

The challenge with managed packages is that the method for creating and updating them is very challenging, involving the use of a separate Developer Edition org, packaging orgs, namespaces, and so on. For these reasons, few enterprises build or use managed packages to migrate functionality between their environments.

The consequence is that although most Salesforce orgs make use of commercial managed packages to extend their org's functionality, homegrown customizations to

[1] www.modulecounts.com/

[2] www.salesforce.com/company/news-press/press-releases/2018/12/181206-s/

orgs are almost never organized into packages. It's common for large enterprises to have tens of thousands of unpackaged pieces of metadata (classes, objects, fields, Flows, etc.) that exist together as an undifferentiated collection.

Some organizations have disciplined themselves to use pseudo-namespace prefixes to distinguish related pieces of functionality, but for the most part it is not easy to see which pieces of metadata are closely related to one another without taking time to inspect their contents.

To address these problems, Salesforce DX introduces two types of second-generation packages: unlocked packages and second-generation managed packages. The former is mostly for use by enterprises, since it does not hide its contents or prevent them from being modified after they've been installed. The latter is a more flexible successor to managed packages, designed for ISVs to be able to deploy and update functionality while still retaining control over most of the contents of those packages.

This book is mostly oriented toward helping enterprises to manage their customizations, and so we discuss unlocked packages at great length. Fortunately, the methods for creating and deploying unlocked packages and second-generation managed packages are almost identical so most of the content in this book is relevant to second-generation managed packages as well.

Metadata API vs. SFDX Source Formats

Another important innovation in Salesforce DX was recognizing that the file format made available by the Metadata API was not conducive to team development in version control. It's common for standard Salesforce objects like the Account object to be used by many applications and extensively customized. Those objects are represented by the Metadata API as XML files that can grow to tens or hundreds of thousands of lines long. Naturally, developers working collaboratively on these objects frequently encounter merge conflicts, issues with sorting tags, and invalid XML.

Salesforce DX brought a new project structure and source file format. Salesforce DX source format uses different file extensions and folder structure compared to the Metadata API format that Salesforce developers are accustomed to. Salesforce DX also provides a new source shape that breaks down large files to make them easier to manage with a version control system. Listing 2-1 shows the traditional Metadata API file structure, while Listing 2-2 shows the equivalent files converted into the "source" format. Note that -meta.xml is used as the file suffix for XML files, and the complex .object files have been decomposed into their subcomponents.

Listing 2-1. The traditional "Metadata API" structure of Salesforce files

```
src
├── applications
│   └── DreamHouse.app
├── layouts
│   ├── Broker__c-Broker Layout.layout
│   └── Property__c-Property Layout.layout
├── objects
│   ├── Bot_Command__c.object
│   ├── Broker__c.object
│   ├── Property_Favorite__c.object
│   └── Property__c.object
└── package.xml
```

Listing 2-2. The new "Source" structure of Salesforce files

```
force-org
└── main
    └── default
        ├── applications
        │   └── DreamHouse.app-meta.xml
        ├── layouts
        │   ├── Broker__c-Broker Layout.layout-meta.xml
        │   └── Property__c-Property Layout.layout-meta.xml
        └── objects
            └── Broker__c
                ├── Broker__c.object-meta.xml
                ├── compactLayouts
                │   └── Broker_Compact.compactLayout-meta.xml
                ├── fields
                │   ├── Email__c.field-meta.xml
                │   ├── Mobile_Phone__c.field-meta.xml
                │   └── Title__c.field-meta.xml
                └── listViews
                    └── All.listView-meta.xml
```

You can read in detail about this format in the Salesforce DX Developer's Guide.[3]

Salesforce Command-Line Interface (CLI)

The Salesforce CLI is a powerful command-line interface that simplifies development and build automation when working with your Salesforce org. Based on the Heroku CLI, the Salesforce team built a flexible, open source CLI engine called OCLIF, the Open CLI Framework. The Salesforce CLI was the first tool built on OCLIF, although other tools have followed.

The Salesforce CLI allows common Salesforce API commands to be called from the command line and also encapsulates complex processes like synchronizing source with a scratch org in concise commands. Importantly it also allows the output from those commands to be exported in JSON format so that it can easily be parsed and possibly passed as input to other commands.

The Salesforce CLI can be used directly on the command line, included in scripts such as CI jobs, and is also the underlying engine powering the Visual Studio Code extensions.

Summary

Salesforce is an extremely powerful and versatile platform. But it's unique in many ways and it hasn't been easy for professional developers to adapt their tools and techniques to working on this platform.

Salesforce DX is a major strategic initiative from Salesforce to ensure that industry best practices such as DevOps and modular architecture are possible on the platform. Salesforce DX includes many components such as scratch orgs, unlocked packages, and the Salesforce CLI. It also encompasses many teams such as the Apex and API teams working behind the scenes to empower these capabilities.

The significance of this shift to DevOps is explained in detail in the next chapter.

[3] https://developer.salesforce.com/docs/atlas.en-us.sfdx_dev.meta/sfdx_dev/sfdx_dev_source_file_format.htm

CHAPTER 3

DevOps

DevOps has become a trending topic in the tech industry over the last decade, gaining the attention of both developers and the C-suite. Despite some legitimate debate over what it does and does not entail, this interest in DevOps has elevated the discussion around software development practices and the impact they can have on how businesses rise and fall. Figure 3-1 gives one indication of the acceleration of interest in the topic of DevOps over time.

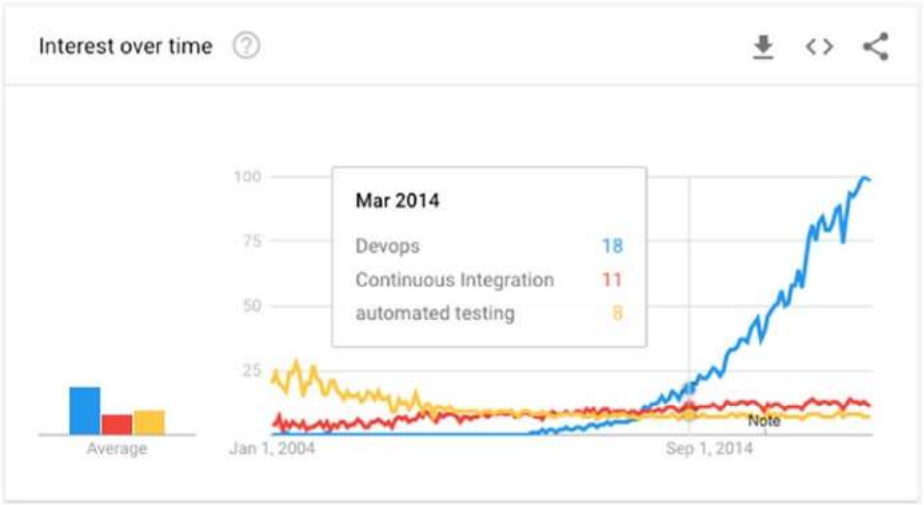

Figure 3-1. *Google Trends ranking showing the acceleration of interest in DevOps compared to related terms*

The practice of DevOps is to regard IT processes as central to a business's ability to deliver value to its customers and to continually improve the process of delivering new functionality while ensuring quality and stability. DevOps is a portmanteau of the terms "Dev" (Development) and "Ops" (Operations). The term implies collaboration between these teams, where they're motivated by the shared goal to enhance both innovation

A. Davis, *Mastering Salesforce DevOps*, https://doi.org/10.1007/978-1-4842-5473-8_3

and stability. Achieving this goal involves integrating long-standing development practices—such as continuous integration, continuous delivery, automated testing, and infrastructure as code. As such, it's become a catch-all term for development process improvements both new and old.

In practice, DevOps is not something you're ever "done with." In that sense it's like "collaboration" or "efficiency"—a principle to adopt and a goal to always improve upon. One core metric used to determine successful implementation of DevOps is called "lead time"—how long does it take your organization to deploy a single line of code to production? It is important that any changes delivered to production be well tested and made in a controlled way. But increased delays in delivering features to production mean greater delays in getting feedback. Delayed feedback means that improvements and bug fixes are delayed, causing inefficient context switching as developers repeatedly revisit work that they may not have touched for days, weeks, or even months.

There are many examples of companies like Amazon, Google, and Etsy who continuously deliver innovation as a result of automating and optimizing their development processes. For companies like these, their development processes are central to their businesses. And their CEOs and other leaders recognize IT process improvement as being every bit as critical as reducing their overhead and building market share. There are also many companies in industries like communication, manufacturing, and banking (Capital One is a notable example), who have shifted to viewing themselves as technology companies, with technological innovation as a core competency.[1]

This chapter provides a brief overview of DevOps, especially for the benefit of Salesforce practitioners who are new to these concepts. There are many excellent books that delve deeply into this topic—among others, *Accelerate* by Nicole Forsgren, *The DevOps Handbook* by Gene Kim, and *Continuous Delivery* by Jez Humble and David Farley.

[1] https://hbr.org/2016/04/you-dont-have-to-be-a-software-company-to-think-like-one

What's Driving the Need for DevOps?

Some companies have been automating and optimizing their code development processes for decades, while others have only recently begun to consider this. Probably the best industry-wide survey of DevOps maturity is the *State of DevOps Report*[2] by Puppet Labs and DORA (DevOps Research and Assessment, now part of Google Cloud). The *State of DevOps Report* shows striking differences between DevOps "haves" and "have-nots." High-performing DevOps teams deploy far more frequently and have vastly shorter cycle times (lead time for changes), fewer failures, and quicker recovery from failures compared to low-performing teams.

At the heart of DevOps is the process of continuous improvement, inspired by the Japanese process of *kaizen*. The implication is that you should get started now, while adopting a discipline of continuous improvement, coupled with the playfulness and flexibility to innovate and experiment. The first and most important step is to capture the state of all of your systems in version control and to perform all your deployments using continuous delivery to ensure that changes are always tracked. These practices set the foundation for increasingly refined automation.

What Is DevOps?

DevOps is a term for a group of concepts that, while not all new, have catalyzed into a movement and are rapidly spreading throughout the technical community. Like any new and popular term, people have somewhat confused and sometimes contradictory impressions of what it is. While a lot of discussions about DevOps focus on the specific tooling or technical implementations, the core meaning relates to the people that develop and maintain technical systems and the culture that surrounds this process. An infinity loop, as shown in Figure 3-2, is often used to illustrate the ongoing rhythm of activities that make up the DevOps process.

[2] https://puppet.com/resources/whitepaper/state-of-devops-report

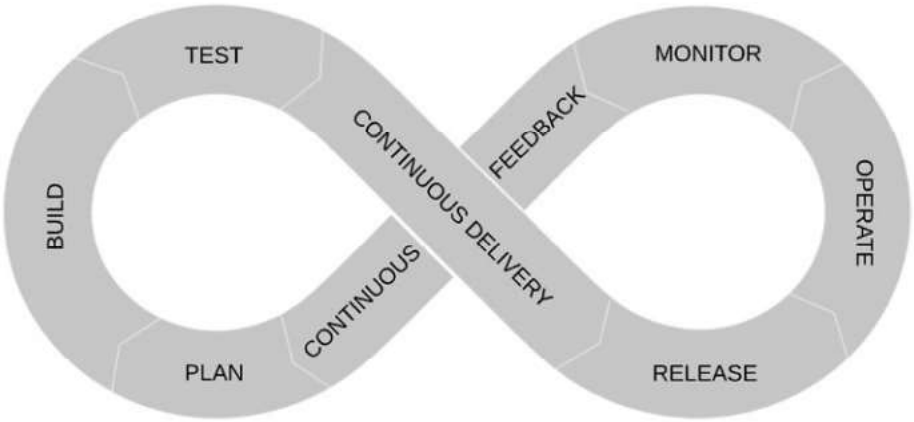

Figure 3-2. *A DevOps infinity loop*

Dev vs. Ops

Working with cloud applications like Salesforce greatly simplifies the requirements for delivering capabilities. In a sense, Salesforce itself takes the place of the "Operations" team that keeps the production system running. Salesforce Admins largely work as declarative developers, building new capabilities rather than just keeping the lights on and keeping performance tuned. But to understand the DevOps challenge, we can think about development and operations needs in a traditional enterprise.

A critical need for businesses is the ability to innovate and develop new functionality for their customers and employees. This is the role of the Development team.

Another critical need is for existing systems to be stable, reliable, and secure. This is the role of the Operations team, which typically consists of system admins, database admins, web site admins, and so on. Their main job is to make sure servers are up and running, SLAs are being met, the application is performing as expected, and so on.

There's a natural tension between the need for innovation (change) and the need for stability. Developers want and need to keep changing the system, while the Operations team wants and needs the system to remain as static as possible so that they can optimize performance and reduce the risk that change brings.

In large organizations, these teams had historically worked in silos which isolated the Development teams from QA and Ops. QA and Ops typically deal with huge numbers of applications and features, sometimes with little understanding of the business purpose and value of the software they were enabling.

The end goal for all of these teams is customer satisfaction, but specific goals for "Devs" include fixing bugs fast and creating new features, whereas for their "Ops" counterparts, the goals might be to maintain 99.99% server uptimes. These goals can often be in conflict with one another, leading to inefficiency and finger-pointing when things go wrong.

Chronic conflict between the Dev and IT operations teams is a recipe for failure for the IT teams as well as the organization they serve.

The concept of DevOps is founded on building a culture of collaboration, communication, and automation between teams that may have historically functioned in silos. The goal is for both Developers and Operations to share responsibility for facilitating innovation while still ensuring stability.

The generative culture promoted in the DevOps community emphasizes values such as ownership and accountability. Developers and operations teams collaborate closely, share many responsibilities, and combine their workflows. This reduces inefficiencies and saves time (e.g., writing code that takes into account the environment in which it is run). Figure 3-3 depicts DevOps as uniting the focus and activities of these previously disparate teams.

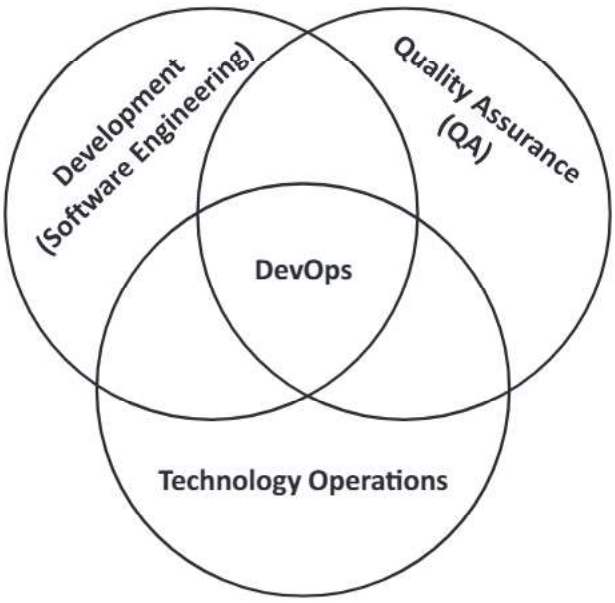

Figure 3-3. *DevOps brings together activities that were traditionally done by independent development, QA, and IT operations teams*

The Real Meaning of Agile

In the book *Accelerate,*[3] the authors make the point that

> *at the time the Agile Manifesto was published in 2001, Extreme Programming (XP) was one of the most popular Agile frameworks. In contrast to Scrum, XP prescribes a number of technical practices such as test-driven development and continuous integration … Many Agile adoptions have treated technical practices as secondary compared to the management and team practices that some Agile frameworks emphasize. Our research shows that technical practices play a vital role in achieving these outcomes.*

The Agile Manifesto itself promotes 12 principles,[4] the first of which is that "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software." And the ninth of which states "Continuous attention to technical excellence and good design enhances agility."

To the extent that most development teams organize their work into sprints, and craft their requirements in the form of user stories, "Agile" has become the dominant practice for delivering software. But if deployments require hours or days of preparation, environments are inconsistent, and development teams are working on differing versions of the codebase, then actual agility necessarily suffers.

Real agility depends on being nimble with changing customer requirements and on promoting and deferring to the evolving understanding of the team. But the systems the team uses to do their work and the underlying architecture of the products they're building are also critically important to achieving real agility.

What better measures of agility could there be than how often the team is able to deploy, how quickly, how often those deployments fail, and how quickly the team can recover?

[3]Nicole Forsgren, Jez Humble, and Gene Kim, *Accelerate: The Science of Lean Software and Devops Building and Scaling High Performing Technology Organizations* (IT Revolution Press, 2018): p. 41.

[4]https://agilemanifesto.org/principles.html

The Three Ways of DevOps

In his books *The Phoenix Project* and *The DevOps Handbook*, Gene Kim popularized the concept of the Three Ways of DevOps. You can find excellent explanations in those books or in blog posts such as this one.[5] The basic idea is depicted in Figure 3-4. I like to summarize these Three Ways as continuous delivery, continuous feedback, and continuous improvement.

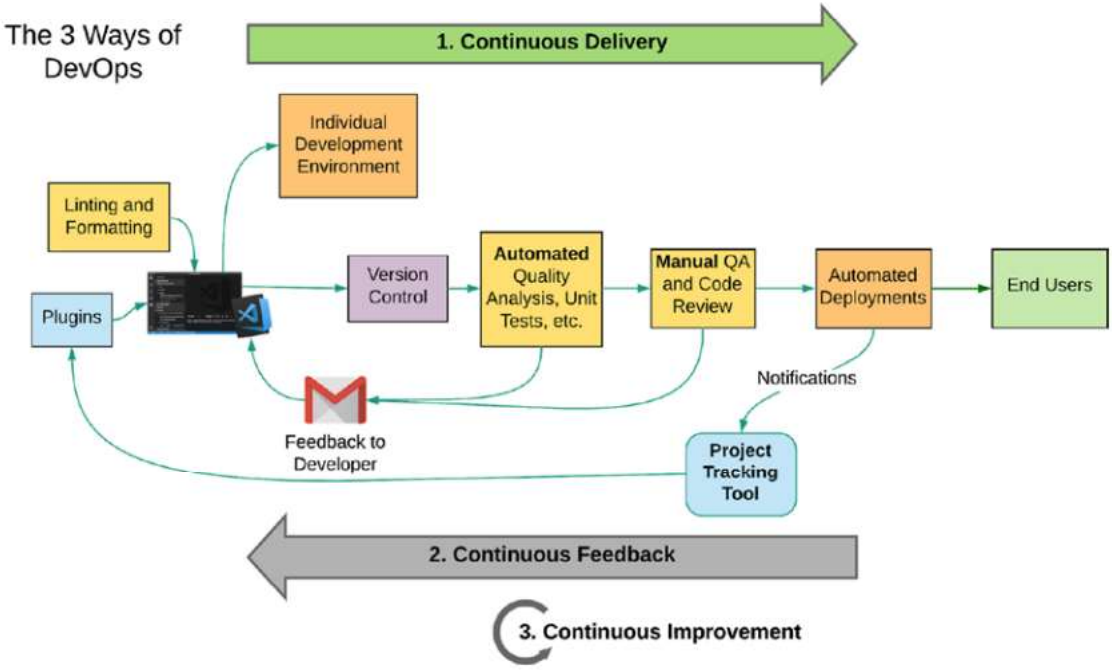

Figure 3-4. *DevOps can be characterized by "Three Ways" of working, illustrated in this diagram*

Continuous Delivery

The First Way is the left-to-right flow of work from Development to QA to IT Operations to Customer. To understand the process of continuous delivery, it helps to understand some basic concepts involved in this process.

[5] https://itrevolution.com/the-three-ways-principles-underpinning-devops/

The foundation for continuous delivery (and thus for implementing DevOps) is the use of version control. Version control is a mechanism to track and merge changes smoothly. There are many types of version control, but the most common type used by developers today is Git. The use of version control provides many benefits, but it also opens the door to all of the practices mentioned later. The reason for this is that code stored in version control can be accessed by automated scripts, with each commit tracked in version control reflecting an iterative improvement that is a candidate for testing and deploying.

Continuous integration means that teams work on a common trunk in the codebase and run automated tests with every commit to that trunk. "CI" is often used to refer to tools such as Jenkins that perform automated actions on a schedule or based on code changes. But the underlying meaning of CI is that no two developers are ever working on a codebase that diverges significantly from one another. CI has been a proven software development practice for decades, in contrast to an approach where teams develop in isolation for weeks, months, or years before eventually entering into a long and painful integration phase.

When using CI, it is typical that all code produced by development teams merges into a central repository, where an automated build validates it. This practice helps development teams detect, identify, and fix problems and bugs before releasing changes to customers. This "CI system" alerts the team to build or test failures, with checks being rerun for even the smallest change, to ensure your system continues to work flawlessly. Early detection and fixes are key to ensuring quality at the earliest possible stage. Developers should focus on setting up a simple continuous integration process as early as possible in the development lifecycle.

Continuous delivery means that all the configuration needed to recreate your application is stored in and deployed from version control in an automated way. It's based on the preceding two practices and is the main technical capability at the heart of DevOps.

As mentioned earlier, DevOps is based on the practice of continuous delivery. DevOps brings together continuous delivery with lean management to maximize the throughput of valuable software and the stability of systems at the same time. That is, "move fast and don't break things!"

Continuous Feedback

The Second Way is the feedback cycle going from right to left; the constant flow of feedback at all stages of the value stream to ensure we can prevent problems from happening again or enable faster detection and recovery. The necessary processes

include the creation of fast automated test suites, monitoring the security, quality, and reliability of the work being passed through the pipelines and failing the pipeline as soon as a test fails, ensuring the code is always in a deployable state.

Continuous Improvement

The continuous delivery and feedback systems described earlier are called the "system of work"—the mechanism that enables valuable work to flow to production. With any changing system, entropy causes things to break down over time, there is no remaining the same. Only by applying effort to continuously improve the "system of work" can it continue to support and empower the whole organization. DevOps demands continual experimentation to improve the "system of work", and to prioritize this system above the work itself. Maturing the DevOps pipeline involves taking risks and learning from success and failure. The necessary practices include creating a culture of innovation, risk taking, and high trust.

Lean Management

DevOps basically combines two movements: the technical practices of continuous delivery and the practice of Lean management and Lean product development that originated with Toyota. The Toyota Production System set the world standard for how to mass produce a top-quality product. And their product development processes set the world standard for how to quickly innovate to satisfy customers' diverse needs. Lean Software Development was born from the realization that applying the same practices to software development increased quality and thereby company performance.

There are two pillars of the Lean approach (whether manufacturing or software): just-in-time and stop-the-line. Just-in-time is focused on speed, throughput, and efficiency. With software, the idea is to minimize the time required to deliver a requested feature through steps like automating deployments and identifying other bottlenecks in the process that don't add value. One key to fast delivery is to break work into small batches. As each small batch is completed, it can be released, which is why the frequency of releases is an excellent indicator of whether teams are following this approach.

Stop-the-line means that as soon as any defect or abnormality is found, the production line (or deployment pipeline) stops and the top priority becomes identifying and uprooting the source of the problem to ensure it doesn't happen again. The focus here is on stability and quality. With software, this is accomplished through building in checkpoints such as automated testing and validations, and rerunning those every time the codebase changes. The idea is to make the system autonomic, like your autonomic nervous system, where your reflexes recoil from fire without your having to consciously think about it. Having a culture of blameless postmortems is a clear foundation for this kind of approach.

The amount of time a customer spends waiting for a feature to be delivered is called Lead Time. What's happening while the customer waits? Three things:

- Valuable work is being done.
- Work is being done that (in retrospect) doesn't add value (and sometimes isn't even necessary).
- The feature is waiting on someone else to build/review/test/deploy it.

It's sometimes hard to tell which actions add value and which actions don't. But it's always the case that time spent waiting doesn't add any value. The fastest sustainable speed at which all of the valuable work could be done is called the Process Time. The difference between the Lead Time and the Process Time is one of the main types of waste.

If you think of the lifecycle of groceries, they're first grown or manufactured, then they're stored or shipped, then eventually they're bought and used. The more time elapses between being grown and being used, the more likely they are to go bad. Software also ages quickly. More importantly, the longer the lead time, the longer it takes to get feedback from actual users. Building software is extremely complex, and the only reliable way to ensure things are built well is to get feedback quickly and repeatedly.

The essence of DevOps (and Lean software development) is to eliminate waste, especially time spent waiting. Reducing lead times allows for fast feedback, fast feedback allows for innovation, and innovation enables success.

One of the easiest ways to eliminate wasted time is to automate your deployments using continuous delivery. Implementing continuous delivery on Salesforce is one of the main focuses of this book.

Generative Culture

In his study of safety cultures, sociologist Ron Westrum famously categorized organizational cultures into pathological, bureaucratic, or generative.[6] Pathological cultures are oriented around gaining and maintaining power. To maintain power, groups and individuals in pathological organizations tend to limit knowledge sharing and retain tight control over their areas of responsibility.

Bureaucratic cultures are rule oriented, with a focus on applying rules consistently across the organization. Bureaucratic cultures have an emphasis on fairness and on following processes.

Generative cultures, by contrast, are performance oriented. The focus in generative cultures is on getting the job done, rather than emphasizing control or process. Control is still important in generative cultures, as are rules and processes; but they are important only in serving the larger mission of organizational effectiveness.

The term DevOps itself indicates collaboration between Dev and Ops. By extension, DevOps involves collaboration with the QA teams, security teams, as well as business owners and subject-matter experts. The origins of DevOps in lean manufacturing imply looking at the entire software delivery chain strategically, to gradually improve the flow of work. The adoption of new development tools and best practices is done as a means to improve the entire system and thus requires the team to work together for the benefit of all.

For these reasons, Westrum's generative culture model is an excellent description of a culture conducive to DevOps. DevOps requires collaboration between different teams as well as learning and experimentation to optimize software delivery. For many companies, the biggest challenge to adopting DevOps is establishing an open, performance-oriented culture that allows for such continuous improvement.

The 2018 State of DevOps survey asked respondents about their team's culture and their organization's culture. The survey analysis concluded that a generative culture is indeed indicative of better company performance. Importantly, it also found that adopting DevOps practices such as continuous delivery actually drives further cultural improvements by reducing the barriers to innovation and collaboration.

---

[6]Ron Westrum, "A typology of Organisational Cultures," *Quality and Safety in Health Care* 13, no. suppl 2 (2004): ii22-ii27.

Blameless Postmortems

One manifestation of such a generative culture is an absence of finger-pointing. Development of a strong "safety culture" is recognized as important across many industries, notably healthcare, manufacturing, and transportation. In the aftermath of any incident (whether a train crash or an outage in an IT system), blameless postmortems should be conducted to determine how such incidents can be averted in future. Rather than ever settling on "human error" as a root cause, emphasis should be placed on the circumstances that allowed that error to occur. Human error itself has root causes, and a rich safety culture builds in protections that allow imperfect humans to nevertheless avoid such risks and dangers.

A powerful example of a blameless postmortem followed a headline-generating outage for Amazon Web Services. AWS has become the computing infrastructure for an enormous number of companies. On February 28, 2017, the S3 storage service in AWS's main region went offline for 4 hours. This took a large number of major Internet companies like Quora and Trello offline during that time, since S3 is used to host web site files among many other things. AWS soon released a postmortem of that event,[7] in which they identified that a typo from one of their admins triggered a chain of other problems.

Although human error was involved, nowhere in their analysis did they cite "human error" as a factor, let alone a root cause. Rather they dug deeper to ask what other circumstances allowed this mistake to unfold. They implemented several new protections and safety checks as a result. Despite the PR problems caused by this incident, there was no implication that the admin responsible was fired or disciplined. Rather energy was invested in further improving AWS to eliminate the risk of this kind of incident happening again in future.

A dramatic example of the long-term impact of improving safety culture can be found by looking at automobile fatalities in the United States over time. Figure 3-5 shows the dramatic increase in the number of miles driven per year, juxtaposed with the progressive reduction in risk of fatalities from automobile travel, along with a few practices that helped bring about these improvements. Blameless postmortems provide the opportunity for IT teams to make similar progress to build systems that are increasingly resilient, even as the velocity of innovation continues to increase.

[7] https://aws.amazon.com/message/41926/

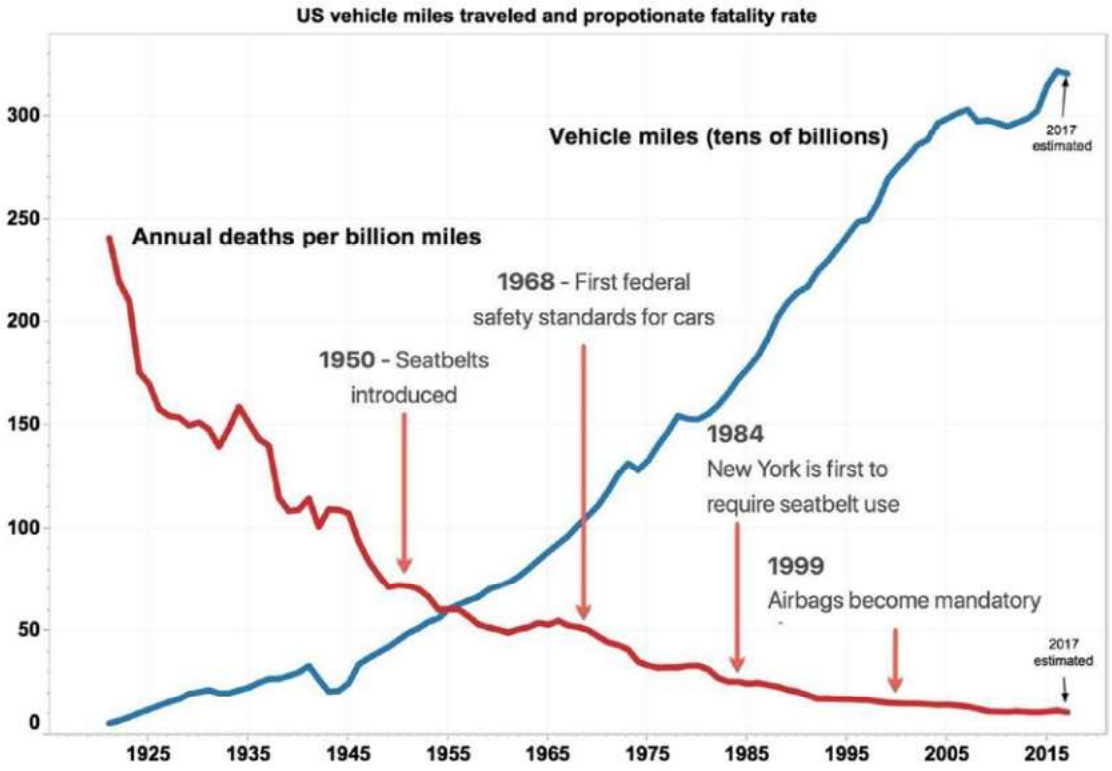

Figure 3-5. *Progress on automobile safety in the United States over time*[8]

The IT industry still has much to learn from the systematic improvements in quality and worker safety that have been implemented through such long-standing safety cultures.

The Research

The use of DevOps (continuous delivery and lean software development) has been shown to improve Software Delivery Performance (how fast you can release functionality in a stable way). According to the *2018 State of DevOps Report*

[8]National Highway Traffic Safety Administration.

> *[Software Delivery and Operational] Performance … enables organizations to leverage software to deliver improved outcomes. These outcomes are measured by many factors, including productivity, profitability, and market share as well as non-commercial measures such as effectiveness, efficiency, and customer satisfaction. Our analysis shows that elite performers are 1.53 times more likely to meet or exceed their goals for organizational performance.*

The *2018 State of DevOps Report* also shows that teams with high Software Delivery Performance (SDP) exceeded those with low SDP with

- 46 times more frequent code deployments
- 2,555 times faster lead time from commit to deploy
- 7 times lower change failure rate (⅕ as likely for a change to fail)
- 2,604 times faster mean time to recover from downtime

The use of continuous delivery (version control, CI/CD, etc.) leads to increased software delivery performance, 38% less rework and unplanned work, and 66% more new (constructive) work. High-performing DevOps teams also have higher employee NPS than low-performing teams.

Business Impact of Adopting DevOps

The *State of DevOps Report* affirms the positive impact that adopting DevOps has on businesses. These studies validate that organizations with high-performing DevOps processes outperform their peers financially by enabling faster time to market for features and increased customer satisfaction, market share, employee productivity and happiness thus allowing them to win in an increasingly competitive economy.

As shown in Table 3-1, organizations like Amazon, Google, Twitter, Facebook, and Etsy embody DevOps processes and routinely deploy hundreds or even thousands of production changes per day while still preserving world-class "reliability, stability, and security." These organizations are more agile, and the time required to go from code committed to successfully running in production is an average of 8,000 times faster. In contrast, organizations that require weeks or months to deploy software are at a significant disadvantage.

Table 3-1. *Software Delivery Performance metrics from some industry-leading software companies*[9]

Company	Deployment Frequency	Deployment Lead Time	Reliability	Customer Responsiveness
Amazon	23,000/day	Minutes	High	High
Google	5,500/day	Minutes	High	High
Netflix	500/day	Minutes	High	High
Facebook	1/day	Hours	High	High
Twitter	3/week	Hours	High	High
Typical Enterprise	Once every 9 months	Months or quarters	Low/medium	Low/medium

Not only do these organizations do more work, they also have far better outcomes. When they deploy code, it is twice as likely to run successfully (i.e., without causing a production outage or service impairment), and when a change fails and results in an incident, the time required to resolve the incident is far faster.

With DevOps in place, instead of deployments being performed only at nights or on weekends, full of stress and chaos, these organizations are deploying code throughout the business day without most people even noticing.

Developers get feedback on their work constantly: linting tools, automated unit, acceptance, integration tests, and other build validations run continually in production-like environments. This gives continuous assurance that the code and environments will work as designed and that code is always in a deployable state.

[9]Gene Kim, Kevin Behr, and George Spafford, *The Phoenix Project: A Novel about It, Devops, and Helping Your Business Win* (IT Revolution Press, 2013), 380.

How DevOps Helps

The *State of DevOps Report* confirms the importance of technical practices such as continuous delivery in improving software delivery performance and thus company performance. But that same survey also speaks to the benefits that come directly to teams adopting these practices. The use of continuous delivery itself drives a positive organizational culture, high job satisfaction, and higher employee engagement.

In addition, the use of continuous delivery (version control, CI/CD, etc.) leads to less rework, less deployment pain, and thus less burnout. This is a virtuous cycle. Practices that benefit the development team also benefit the customer; and happier customers in turn make the development team more engaged.

Better Value, Faster, Safer, Happier

Jonathan Smart, who led Barclays Bank on a journey of enterprise-wide DevOps transformation, summarized DevOps as "Better value, faster, safer, happier."[10]

Better means always striving to improve the quality of our work and our products. This implies continuous improvement.

Value means using agile development and design thinking to address and adapt to the needs of end users. The focus here is on delivering the desired results to end users and improving their experience. Software is not "done" until it's in the hands of users, the ones who actually experience value from it.

Faster means using techniques such as continuous delivery to release more quickly. Speed matters; life is short. The more time elapses between requesting something and getting it, the less time that solution will remain valuable. The more time between creating something and getting feedback on it, the more time is wasted in waiting and inevitably forgetting how to improve it.

Coupling increasingly frequent releases with increasingly automated testing allows you to innovate for customers faster, adapt to changing markets better, and improve your product faster. The quicker you can release new features and fix bugs, the faster you can respond to your customers' needs and build competitive advantage.

[10] https://medium.com/@jonathansmart1/want-to-do-an-agile-transformation-dont-focus-on-flow-quality-happiness-safety-and-value-11e01ee8f8f3

Safer means focusing on stability and integrating automated quality and security scanning into the development workflow. Just as when driving in a car, speed only matters if you can arrive safely. Safety means that the solutions we deliver will work and that they won't break other things. Safety implies testing, or quality assurance. Testing means testing that the software does what it's supposed to do, is resilient against changing conditions, and doesn't introduce regression failures (break things that used to work). Safer also implies security, which is an aspect of structural quality.

Automated testing gives confidence that each change is functional and safe. Monitoring and logging practices help teams stay informed of performance or issues in real time.

And **happier** refers to continuously improving the development experience and the customer experience of users. We're humans developing for humans. The experience should be increasingly positive for the developers creating functionality and increasingly positive for the end users consuming it.

Note that all of these terms are relative and subjective. This implies an ongoing process, where value is flowing from developers to end users (from creators to consumers), and the whole process is improving on an ongoing basis. We all want better and better experiences. Wisdom dictates that if we focus on improving our actions, our experiences will naturally improve. And so wisely, we strive to improve ourselves and our teams, so that together we can do a better and better job.

The team with the fastest feedback loop is the team that thrives. Full transparency and seamless communication enable DevOps teams to minimize downtime and resolve issues faster than ever before.

Measuring Performance

One of the main contributions of the *State of DevOps Report* has been to focus consistently on the same key metrics year after year. Although the questions in their survey have evolved and new conclusions have emerged over time, the four key metrics used as benchmarks have remained in place:

1. Lead time (from code committed to code deployed)
2. Deployment frequency (to production)
3. Change Fail Percentage (for production deployments)
4. Mean Time to Restore (from a production failure)

The book *Accelerate* provides a detailed explanation of each of these metrics and why they were chosen; those points are summarized here.

The first two of these metrics pertain to innovation and the fast release of new capabilities. The third and fourth metrics pertain to stability and the reduction of defects and downtime. As such, these metrics align with the dual goals of DevOps, to "move fast, and not break things."

These also align with the two core principles of Lean management, derived from the Toyota Production System: "just-in-time" and "stop-the-line." As mentioned earlier, just-in-time is the principle that maximum efficiency comes from reducing waste in the system of work and that the way to reduce waste is to optimize the system to handle smaller and smaller batches and to deliver them with increasing speed. "Stop-the-line" means that the system of work is tuned not just to expedite delivery but also to immediately identify defects to prevent them from being released, thus increasing the quality of the product and reducing the likelihood of production failures.

Lead Time is important because the shorter the lead time, the more quickly feedback can be received on the software, and thus the faster innovation and improvements can be released. *Accelerate* shares that one challenge in measuring Lead Time is that it consists of two parts: time to develop a feature and time to deliver it. The time to develop a feature begins from the moment a feature is requested, but there are some legitimate reasons why a feature might be deprioritized and remain in a product's backlog for months or years. There is a high inherent variability in the amount of time it takes to go from "feature requested" to "feature developed." Thus Lead Time in the *State of DevOps Report* focuses on measuring only the time to deliver a feature once it has been developed. The software delivery part of the lifecycle is an important part of the total lead time and is also much more consistent. By measuring the Lead Time from code committed to code deployed, you can begin to experiment with process improvements that will reduce waiting and inefficiency and thus enable faster feedback.

Deployment frequency is the frequency of how often code or configuration changes are deployed to production. Deployment frequency is important since it is inversely related to batch size. Teams that deploy to production once per month necessarily deploy a larger batch of changes in each deployment than teams who deploy once per week. All changes are not created equal. Within any batch of changes, there will be some which are extremely valuable, and others that are almost insignificant. Large batch sizes imply that valuable features are waiting in line with all the other changes,

thus delaying the delivery of value and benefit. Large batches also increase the risk of deployment failures and make it much harder to diagnose which of the many changes was responsible if a failure occurs. Teams naturally tend to batch changes together when deployments are painful and tedious. By measuring deployment frequency, you can track your team's progress as you work on making deployments less painful and enabling smaller batch sizes.

Change Fail Percentage measures how frequently a deployment to production fails. Failure here means that a deployment causes a system outage or degradation or requires a subsequent hotfix or rollback. Modern software systems are complex, fast-changing systems, so some amount of failure is inevitable. Traditionally it's been felt that there's a tradeoff between frequency of changes and stability of systems, but the highly effective teams identified in the *State of DevOps Report* are characterized by both a high rate of innovation and a low rate of failures. Measuring failure rate allows the team to track and tune their processes to ensure that their testing processes weed out most failures before they occur.

Mean Time to Restore (MTTR) is closely related to the Lead Time to release features. In effect, teams that can quickly release features can also quickly release patches. Time to Restore indicates the amount of time that a production system remains down, in a degraded state, or with nonworking functionality. Such incidents are typically stressful situations and often have financial implications. Resolving such incidents quickly is a key priority for operations teams. Measuring this metric allows your team to set a baseline on time to recover and to work to resolve incidents with increasing speed.

The *2018 State of DevOps Report* added a fifth metric, System Uptime, which is inversely related to how much time teams spend recovering from failures. The System Uptime metric is an important addition for several reasons. First of all, it aligns with the traditional priorities and key performance indicators of sysadmins (the operations team). The number one goal of sysadmins is Keeping the Lights On or ensuring that systems remain available. The reason for this is simple: the business depends on these systems, and when the systems go down, the business goes down. Outages are expensive.

Tracking System Uptime is also central to the discipline of Site Reliability Engineering (SRE). SRE is the evolution of the traditional sysadmin role, expanded to encompass "web-scale" or "cloud-scale" systems where one engineer might be responsible for managing 10,000 servers. SRE emerged from Google, who shared their

practices in the influential book *Site Reliability Engineering.*[11] One innovation shared in that book is the concept of an "error budget,"[12] which is the recognition that there is a tradeoff between reliability and innovation and that there are acceptable levels of downtime.

> *Put simply, a user on a 99% reliable smartphone cannot tell the difference between 99.99% and 99.999% service reliability! With this in mind, rather than simply maximizing uptime, Site Reliability Engineering seeks to balance the risk of unavailability with the goals of rapid innovation and efficient service operations, so that users' overall happiness—with features, service, and performance—is optimized.*[13]

The *State of DevOps Report* shows how these five metrics are interrelated, illustrated in Figure 3-6. The timer starts on Lead Time the moment a developer finishes and commits a feature to version control. How quickly that feature is released depends on the team's deployment frequency. While frequent deployments are key to fast innovation, they also increase the risk of failures in production. Change Fail Percentage measures this risk, although frequent small deployments tend to reduce the risk of any given change. If a change fails, the key issue is then the Mean Time to Restore service. The final metric on availability captures the net stability of the production system.

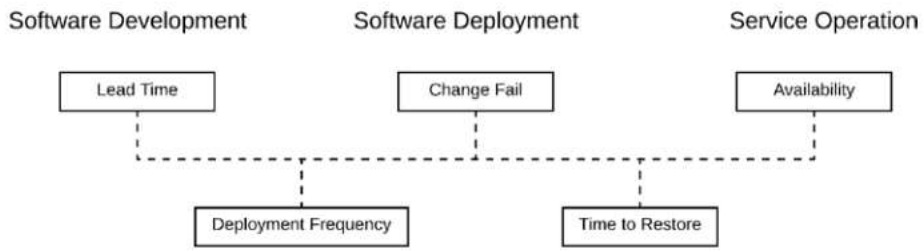

Figure 3-6. *How the five key software delivery and operations performance metrics tie together*

[11] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy, *Site Reliability Engineering: How Google Runs Production Systems* (O'Reilly Media, 2016).

[12] https://landing.google.com/sre/sre-book/chapters/embracing-risk/

[13] *Site Reliability Engineering*, Chapter 3.

Together, these metrics constitute a team's **Software Delivery Performance**. The goal of any DevOps initiative should be to improve Software Delivery Performance by strategically developing specific capabilities such as continuous delivery and the use of automated testing.

How your team measures these capabilities is another challenge. But *Accelerate* makes a compelling argument for the validity of surveys. Automated metrics can be implemented over time, although the mechanism to do this will depend on how you do your deployments. Salesforce production orgs track past deployments, but it's not currently possible to query those deployments, and so you would need to measure deployment frequency (for example) using the tools you use to perform the deployments. Salesforce publishes their own service uptime on `https://trust.salesforce.com`, but that gives no indication of whether critical custom services that your team has built are in a working state or not.

Surveys provide a reasonable proxy for these metrics, especially if responses are given by members of the team in different roles. Guidelines for administering such surveys are beyond the scope of this book, but your teams' honest responses are the most critical factor. Avoid any policies that could incent the team to exaggerate their answers up or down. Never use these surveys to reward or punish; they should be used simply to inform. Allow teams to track their own progress and to challenge themselves to improve for their own benefit and for the benefit of the organization. As it says in the *Agile Manifesto,* "At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly."

Enhancing Performance

High software delivery performance is associated with objective improvements in corporate performance (both commercial and noncommercial) and subjective improvements to deployment pain, burnout, and employee satisfaction. It's therefore in the best interest of everyone in an organization to strive to improve their software delivery performance by monitoring the metrics mentioned earlier.

What methods are available to improve software delivery performance? The research from the *State of DevOps Report* has identified 32 factors that drive this performance. The book *Accelerate* categorizes those factors into five groups:

1. Continuous delivery
2. Architecture

3. Product and process
4. Lean management and monitoring
5. Cultural

In this book, we'll focus almost entirely on how to implement technical practices related to architecture and continuous delivery, with some references to monitoring as well. As such, our goal here is not to provide a comprehensive prescription for all the cultural, management, and process improvements that might be needed, but to at least provide more guidance and definition on how a high-functioning Salesforce implementation would be architected, deployed, monitored, and maintained.

This chapter provides a brief summary of the cultural and lean management practices necessary for success, but for further guidance the reader is encouraged to look at the many excellent books, articles, and talks that are available on these topics.

Optimizing the Value Stream

A fundamental concept in Lean software management is to understand the software development and delivery process in terms of how each aspect of the process contributes (or does not contribute) to value for the end user. As mentioned earlier, value reflects the benefit that the end user receives and would be willing to pay for. As software moves through the development and delivery process, different individuals and teams add value through their contributions. Their ongoing contributions to the process are described as flowing into a "value stream" that delivers ongoing value to end users.

Value Stream Mapping is a key tool in lean management. This involves mapping each stage of the development and delivery process and then gathering three metrics from each stage: lead time, process time, and percent complete and accurate. Lead time is the total amount of time that it takes from a work item entering that stage until it leaves that stage. Process time is the amount of time that would be required for that activity if it could be performed without interruption. And percent complete and accurate is the percentage of the output from that stage that can be used, as is, without requiring rework or clarification. The goal of this mapping is to identify slowdowns where work items spend time waiting or could be completed more efficiently, as well as stages that suffer from quality issues. It's also possible that this process uncovers steps that don't add

much value and can be eliminated or simplified. The core concept of lean management is to assess this process and identify how to eliminate waste from this system, to maximize the flow of value.

The potential technical improvements that can be made to a software product or to the way of working on that software product are limitless. This book alone introduces a huge range of possible improvements to the way code is written or developed, the way it's tested, packaged, deployed, and monitored. But it is unwise to assume that adopting DevOps means that we should simultaneously take on all of these improvements or make improvements arbitrarily. Instead, the goal should be to identify the bottlenecks in the system that limit the overall flow of value and to optimize the system around those bottlenecks.

Begin by identifying the current state value map, and then craft a future state value map that aims to increase the percent complete and accurate of the final product or reduce the amount of waste (lead time where no process is being performed). For more guidance, see the book *Lean Enterprise: How High Performance Organizations Innovate at Scale*[14] or *Value Stream Mapping: How to Visualize Work and Align Leadership for Organizational Transformation*.[15]

It can be very challenging to discern where to begin such a process. One complementary approach known as the *theory of constraints* can provide a practical simplification to help target priority areas for improvement.

[14]Jez Humble, Joanne Molesky, and Barry O'Reilly, *Lean Enterprise: How High Performance Organizations Innovate at Scale* (O'Reilly Media, 2015).

[15]Karen Martin and Mike Osterling, *Value Stream Mapping: How to Visualize Work and Align Leadership for Organizational Transformation* (McGraw-Hill, 2013).

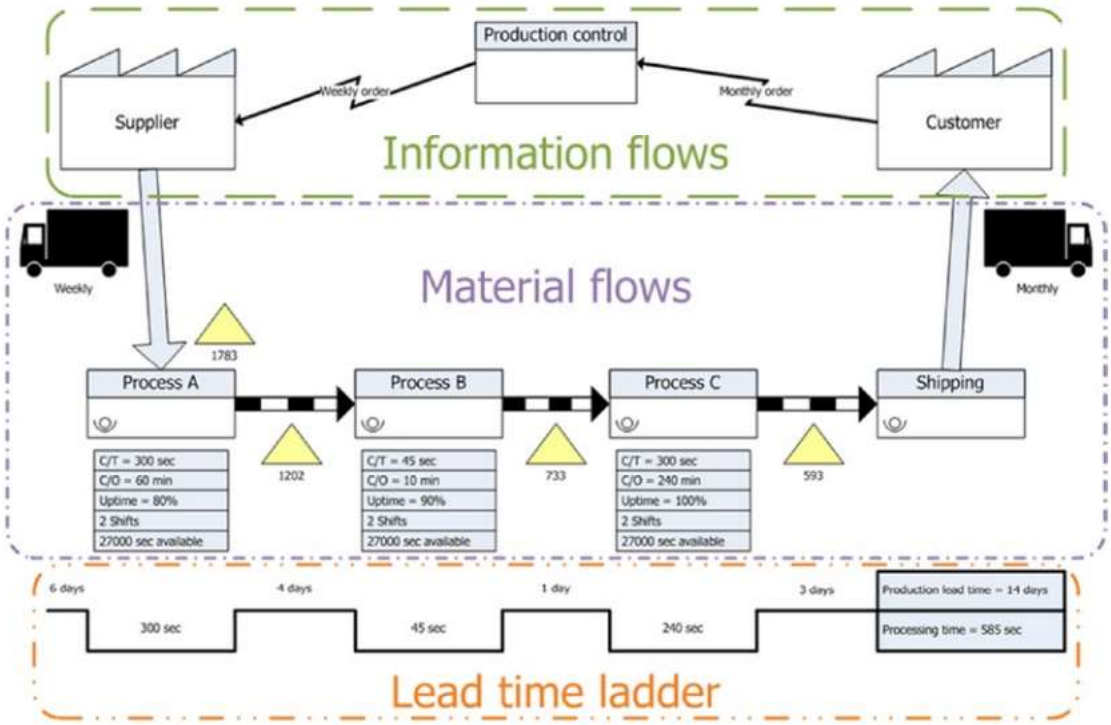

Figure 3-7. *A sample value stream map*

Theory of Constraints

The concept known as the theory of constraints was introduced and popularized by Eliyahu M. Goldratt in his book *The Goal.*[16] That book is cited extensively in DevOps literature and emphasizes a fourfold approach to optimizing the flow of value through a system:

1. Identify the constraint
2. Exploit the constraint
3. Subordinate and synchronize to the constraint
4. Elevate the performance of the constraint

[16]Eliyahu M Goldratt and Jeff Cox, *The goal: a process of ongoing improvement* (North River Press, 2004).

In layman's terms, the idea is that the overall performance of any system at any given time will always be limited by a single constraint. In a software development process, the constraint might be that there are not enough developers, or that the testing process takes too long, or that the deployment process can only be done at certain times, or that there is not any additional demand from customers, and so on. The most important point to note is that **improvements to any part of the system other than this constraint will not yield any significant benefit**. For this reason, the most critical step to making overall improvements is to first identify the constraint.

Identifying the constraint on a system can be a challenging process, combining measurement, intuition, and experimentation. One approach to identifying the constraint is to look for a buildup of "inventory" in a system, since this tends to indicate a point where the process slows down. The concept of inventory is borrowed from manufacturing, but can be extended to the software development process and relabeled as "work in progress." Where does work in progress (WIP) accumulate in your development process? You can ask your team questions like how large is the backlog for developers? How many pieces of work are waiting for review by a tech lead? How much work is waiting to be tested? How many items are waiting to be deployed? Asking questions like this and observing trends over time can help you home in on the main constraint in your process.

Once the constraint has been identified, the biggest improvement that can be made is to **exploit that constraint**—to make sure that it is fully utilized. At the time that Goldratt wrote *The Goal*, it was common in manufacturing to focus on getting maximum utilization from every single system involved in a manufacturing process. The equivalent in a software team is ensuring that everyone is busy all the time. But in practice this local optimization of each part of the system *does not* optimize the performance of the overall system, because overall performance is always defined by a single constraint. It is that constraint that needs to be optimized, getting the maximum sustainable productivity out of that constraint.

The third stage is to **subordinate and synchronize to the constraint**. This means that every other part of the system should be seen as a means to enable and support the constraint. The priority for other systems should be to ensure that the constraint is never waiting on "raw materials" (such as specifications for work) and that the work produced by the constraint is quickly whisked away to the next phase of the process.

For example, if you determine that the capacity of the development team is the limiting factor on your ability to deliver value, then the other individuals or processes that contribute to the value stream—business analysis, architecture, testing, and deployment—should ensure that the developers can be made as effective as possible. That means ensuring that they have a backlog of work and clear architectural guidance, and that once their work is complete, it can be tested and deployed as quickly as possible. Importantly, releasing quickly also allows bugs to be identified quickly so that developers can address those while the relevant code is still fresh in their minds.

Having identified the constraint, the second and third stages of optimization function to maximize the throughput of the constraint (first by using it fully and then by organizing everything else around it). It is entirely possible that through making these optimizations, the original constraint ceases to be the constraint. By definition, the constraints on a system are subject to moving and changing as conditions evolve. This is why it's important to begin by mapping the entire value stream and continuing to monitor each component of it to gain insight into your overall flow. If you find that the constraint has changed, then you have just achieved the first level of optimization for that new constraint: identifying it. Your task then becomes how to exploit the new constraint, and so forth.

If a constraint persists despite your maximizing its throughput, you have only one remaining option: to **elevate the performance of that constraint**. When your constraint is one or more individuals, elevating the performance of that constraint might take the form of giving them better tools, training or coaching them, hiring additional team members, or even replacing them and moving them to a different role if necessary.

This fourfold process is a continual dance. There is never a time when there is not some constraint on a system. Even when the market itself is your constraint, your practice is the same: to exploit that market, subordinate your other activities to feeding market demand, and finally work to elevate demand through marketing and publicity.

Even the optimization process itself is subject to unseen constraints. It takes time, effort, and situational awareness to understand how your value stream is performing; and it can take time, effort, and money to make the changes necessary to improve that performance.

From the point of view of the theory of constraints, this continual dance is the essence of effective management.

Enabling Change

Suffice it to say, implementing or improving any of these capabilities is a change management process that relies on human communication, learning, and experimentation.

Industry surveys conducted by McKinsey[17] indicate that only 25% of transformation initiatives have been very or completely successful; however, organizations that take multiple parallel actions to improve and especially those which emphasize communication have a 79% success rate. In any case, the DevOps journey is not one that is ever "done." The point is to provide a clear vision for your team about areas for possible improvement; to provide training, examples, and encouragement to help them understand how to proceed; and to proceed systematically and incrementally with a process of continuous improvement.

It's helpful to consider the *law of diffusion of innovation*, first introduced by Everett Rogers in 1962[18] and concisely summarized in this diagram. According to this widely cited model, individuals adopt a new process (such as DevOps or Salesforce DX in this case) at different rates. A small few individuals fall into the Innovators category. They are the ones who initially experiment with a technology and lay the early foundations for others to follow. They are followed by early adopters, who tend to copy more than innovate, but who are nevertheless bold and enthusiastic about taking on this new process. Innovators and early adopters may only account for 16% of a total population. But they are truly trailblazers and open the door for subsequent adoption. At these early stages, it may seem that adoption levels are very low, which can be disheartening for those trying to champion change. But these initial stages of adoption are the foundation that eventually leads to a tipping point as the early majority and late majority begin to follow suit. The "S" shaped curve in this diagram shows total adoption and depicts how the slow early phases give way to a period of fast diffusion as the majority gets on board.

[17] www.mckinsey.com/business-functions/organization/our-insights/how-to-beat-the-transformation-odds

[18] Everett Rogers, *Diffusion of Innovations*, 5th Edition (Simon and Schuster, 2003).

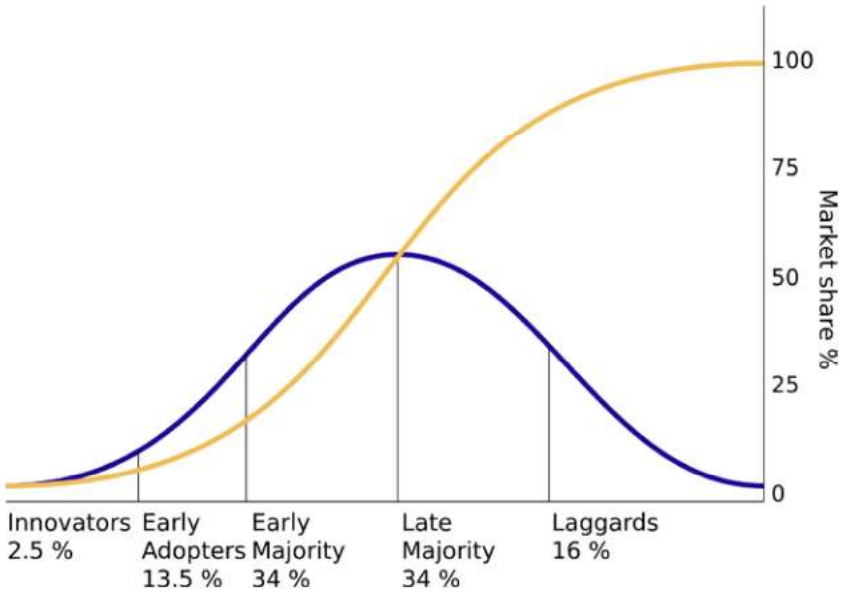

Figure 3-8. *An illustration of the law of diffusion of innovation. Eagerness to adopt a new technology or practice typically follows a bell curve distribution across a population. The cumulative increase in adoption follows an S curve (the integral of the bell curve)*[19]

This law of diffusion of innovation has been the basis for many other analyses. Two relevant observations that are frequently made are that there is a tipping point in this system at which adoption becomes self-sustaining, but also that there is a chasm that needs to be crossed when you transition to majority adoption.

First, the good news. Once you have gained adoption from the innovators and the early adopters (the initial 16% of your target population), you have reached a tipping point where adoption can become self-sustaining, as shown in Figure 3-9. As the early and late majority begin to adopt these processes, the momentum of word-of-mouth marketing and other network effects begins to take hold. People begin to follow this process because it starts to become standard practice. Their friends are doing it; other teams are doing it; the team they used to be on did it, and so on. Malcolm Gladwell's bestselling book *The Tipping Point*[20] was based partly on these same ideas.

[19]Image source: https://commons.wikimedia.org/wiki/File:Diffusion_of_ideas.svg

[20]Malcolm Gladwell, *The Tipping Point: How Little Things Can Make a Big Difference* (Back Bay Books, 2002).

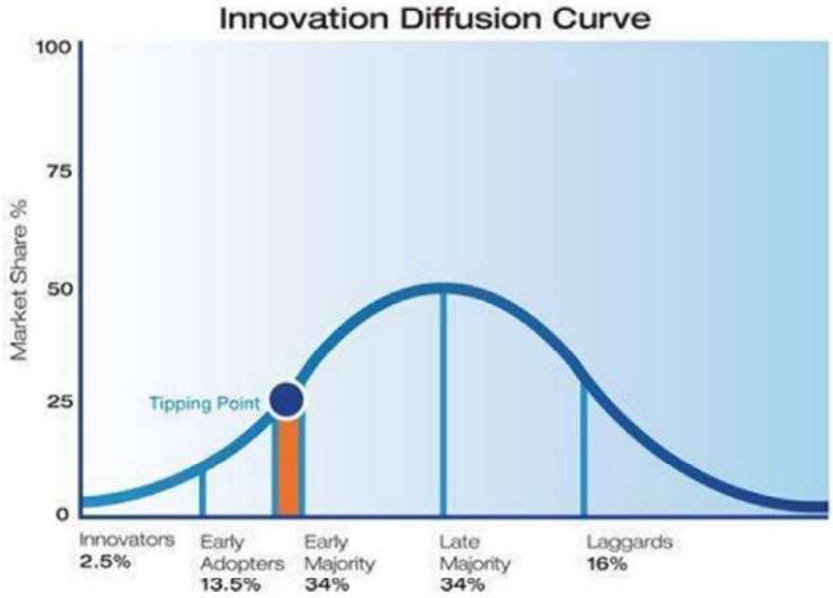

Figure 3-9. *As the early majority begins to embrace an innovation, adoption reaches a tipping point where further adoption tends to become self-sustaining*[21]

Now for the bad news. While there will always be people who explore and experiment with new approaches, Geoffrey Moore pointed out in *Crossing the Chasm*[22] that for new and disruptive technologies, there is a chasm that exists between the early adopters and majority adoption, illustrated in Figure 3-10. Many very promising technologies have achieved adoption from a small corps of aficionados, but never managed to get wide market adoption. Moore explains that this is because there is a markedly different psychodynamic between innovators and the bulk of the population. To cross this chasm, Moore argues, you must reconsider your target market for each stage and tailor your marketing message and medium differently when appealing to the majority.

[21]Source: (CC) Gavin Llewellyn, www.flickr.com/photos/gavinjllewellyn/6353463087

[22]Geoffrey A. Moore, *Crossing the Chasm: Marketing and Selling Technology Products to Mainstream Customers*. (HarperBusiness, 1991).

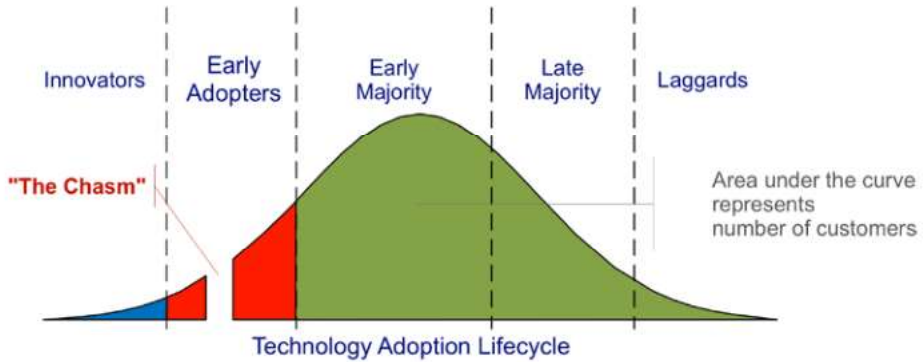

Figure 3-10. *Early adoption of an innovation is driven by different motivators than later adoption. This leads to a "chasm" that often prevents adoption by the majority. The messaging and enablers for an innovation need to change if it is to appeal to the majority*[23]

These ideas are equally applicable for the commercial adoption of a technology or for internal change enablement with one important difference. Inside an organization there is always the possibility of making a change mandatory, tracking and rewarding based on adoption metrics, and so on. As much as they might wish for it, such options are not available to those marketing commercial products!

Nevertheless, effective change inside an organization is most powerful and sustainable when it's driven by intrinsic motivators—when people feel that the new process truly benefits them and their team. To take this organic approach, at the beginning, you should seek to attract and appeal to potential DevOps innovators within your company, working to enable them. As you get adoption from this initial group, you can gradually shift your focus to gaining broader adoption by considering different strategies as you begin to roll out change across more and more teams.

DevOps in general and Salesforce DX in particular represent a paradigm shift for Salesforce developers. There are significant changes to behavior that are required for teams to shift from making and deploying changes in an ad hoc or manual way to tracking and automating this process. Those charged with leading change in their organization should take heart and emphasize getting buy-in from innovators and early adopters while encouraging them to share their experience with others around them. They then become the basis on which you can begin to work toward majority adoption,

[23]Figure source: https://upload.wikimedia.org/wikipedia/commons/d/d3/Technology-Adoption-Lifecycle.png

amplifying those early success stories and building a network of support that can help these changes take hold in the rest of your company.

Leading Change

If you are in charge of a small team of Salesforce developers, you may be able to lead adoption of Salesforce DX in a simple and organic way just by piloting its usage and demonstrating some of its benefits. But if change needs to happen across a large and distributed organization, you have entered into the realm of organizational change management. If you have access to experts in organizational change, you should lean on them for advice on succeeding at this process.

A leading voice in organizational change management is John Kotter, whose book *Leading Change*[24] has been an important and influential guide for many organizational initiatives. Kotter defines an eight-stage process by which effective change takes place. The eight-stage process for leading effective change[25] is shared here, together with some notes on the relevance to improving DevOps processes.

Step 1: Create Urgency

None of us act unless there's a need to. Especially when it comes to changing habitual behaviors and learning new practices, the motivation for change must be strong to overcome our inertia. The same holds true for organizations. Organizational change derives from a sense of urgency, and the first step in effecting broad change is to stimulate this sense of urgency in your leadership team.

Kotter suggests that for change to be successful, 75% of a company's management needs to "buy into" the change. When it comes to DevOps, this mostly pertains to the managers responsible for overseeing development teams. But changes in process can sometimes involve temporary slowdowns as a team learns new practices.[26] There needs to be a commitment from anyone who is a stakeholder in the development process, including project managers, internal customers, and even external customers if you are a consulting company doing development at their behest.

---

[24]John P Kotter, *Leading Change* (Harvard Business Review Press, 2012).

[25]See `www.mindtools.com/pages/article/newPPM_82.htm` for another explanation of this process.

[26]This is sometimes called a "J curve" or the valley of despair. Things get a bit worse before they can get much better.

Urgency is built from understanding the problems implicit in the current approach and the benefits of adopting a new approach. The *State of DevOps Report* provides compelling statistics on the impact of adopting DevOps practices, and I've touted those statistics hundreds of times to thousands of listeners to help gain buy-in and motivation.

It's important for you to also identify risks and opportunities that are specific to your company. In Chapter 12: Making It Better, I share Lex Williams' story of Salesforce's own DevOps adoption. Their technical team had been arguing for years that production changes should only be made through a continuous delivery system. But it was only once they started documenting the financial cost of outages arising from ad hoc production changes that they managed to get executive approval to adopt that kind of governance.

Begin to track metrics on how long it takes your teams to deploy to production, merge codebases, perform regression tests, and so on. Take note of how often one team is prevented from deploying critical changes because they have to wait on other teams to complete their work. Gather statistics and explanations to support your vague sense that processes could be more efficient or effective than they currently are. These arguments form the basis for convincing other stakeholders of the need for change.

At the same time, these analyses need to be honest, and you need to listen to counterpoints and have dialog around the need for change as well as possible risks. There are indeed risks associated with change. But by discussing costs and benefits clearly and honestly, you can help yourself and others discern when the risks of not making these improvements outweigh the risks of making them.

Step 2: Form a Powerful Coalition

The first step is the most important, and you should spend time to make sure you have won organizational support before proceeding. But buy-in from management is only the beginning. You now need to gather others who can act as change agents within the company.

Change takes leadership. But leadership happens in many ways, at many levels. You need to find influential people across your organization who are similarly committed to seeing these changes take root. These people may or may not hold traditional leadership roles. Their influence may instead derive from their length of time at the company, their technical expertise, their passion, or their social connections.

Connect with these people to form a coalition. Ask them to commit to helping bring about this change, and work on building trust and collaboration within this initial group. It is helpful to ensure that you have a broad mix of people from different roles and different groups within your organization. For example, at Appirio, we began our DX initiative with a survey of all our technical consultants in which we asked them "which Appirian has helped you the most to improve your coding skills." This survey helped us to identify influencers across different levels and geographic regions. We brought these people together as evangelists, giving them early access to training and demos as we began to promote Salesforce DX.

Step 3: Create a Vision for Change

Humans are remarkably good at telling and remembering stories. And the most effective communicators rely on stories to convey the essence of complex messages in memorable and impactful ways. Salesforce has poured a phenomenal amount of money into their sales and marketing efforts; and marketing is fundamentally about storytelling.

The tagline for Salesforce DX is "Build together and deliver continuously with a modern Salesforce developer experience."[27] That succinct statement creates a vision, which itself inspires change. "Build together" speaks to collaboration and promises easy collaborative development even across large and distributed teams. "Deliver continuously" speaks to easing deployments and helping end users get ongoing value from development. "A modern developer experience" speaks to the promise of replacing tedious manual processes with powerful and automated tools.

What is the vision for modernizing your own Salesforce development experience?

Think about the values that are most important for this initiative. Are you most concerned about speed? Security? Reliability? Ease of debugging?

Based on these values, write out a one- or two-sentence summary of what the future looks like once these changes are in place. This becomes the vision statement for your change.

[27] https://developer.salesforce.com/platform/dx

Next craft a strategy to achieve that vision. The strategy adds detail that you'll need as you begin to roll out changes. But the first step in your strategy is to regularly reinforce the vision and ensure that everyone in your change coalition is able to articulate this vision and some key aspects of the strategy. This is the message that you need to build and amplify. It needs to be clear, simple, and compelling so that the story can be told, remembered, and retold again and again throughout the organization.

Step 4: Communicate the Vision

Having gathered support from management, built a coalition of change agents, and established a clear and compelling vision for change, you now need to communicate that vision repeatedly across many channels.

People are subjected to many messages and stories every day. In a busy company, even attendance at ongoing training or all-hands meetings may be limited. So for a message to gain traction, it needs to be heard repeatedly, across different mediums, over a prolonged period. At one point in the Appirio DX initiative, I recognized that many people in our organization lacked grounding in some basic DevOps concepts. So I initiated a "DevOps December" campaign to reinforce some basic DevOps ideas across many channels.

We began the campaign somewhat quietly with posts on our internal Chatter group. But my manager encouraged me that we should reiterate these messages across different channels. "I want people to have DevOps coming out of their ears. I want them to hear about this so much that I start getting complaints!" he said. I happily obliged and sent out daily "byte-sized" updates by both email and Chatter, along with an appearance on a company all-hands call, several open webinars, and a dozen small group meetings. Eventually, he did receive complaints, so we limited the volume and distribution a bit. But we were both delighted with the impact and the reach. No one at the company escaped that December without learning a bit about DevOps.

This formal messaging is not the only way to reinforce the vision. Your actions speak louder than words. So embed these concepts into how you work. Call out teams who are adopting parts of these practices and highlight the benefit they're receiving. Invite people from those teams to talk about their process. Look for examples in other technologies and share those to bring inspiration.

Tie everything back to the vision. And look for opportunities to incent people to move toward this goal, rewarding them when they do.

Step 5: Remove Obstacles

Repeated communication represents the "happy path" of sharing the vision for change. But it's entirely natural for you to encounter obstacles as you move toward this goal. Those obstacles may be in the form of skeptics or naysayers who emphasize the shortcomings with the new process or downplay the need for change. There may also be processes or structures in place that get in the way of change.

I've encountered senior technical architects who have said that it will never be possible for us to get all of our teams using version control, or that Salesforce DX is not mature enough or a high enough priority for most teams to adopt. Such voices can represent healthy skepticism, but in many cases they are simply based on these individuals' own inertia or lack of experience with this new way of working.

Listen to skeptics, and make sure you're understanding the doubts and risks being expressed. But be on the lookout for voices and processes that are actively standing in the way of change and figure out how to overcome these. Help educate these skeptics if their concerns are not valid. Or help them understand how the benefits of change outweigh the disadvantages. There is a learning curve to adopting version control. But every developer I've talked to who has worked extensively with version control and continuous delivery would never go back to their old way of working and would want these systems in place on any project they were on in future.

If the obstacles relate to time challenges, the need for training, gaps in tooling, and so on then address those obstacles appropriately.

Kotter's eight steps to effective change were originally based on his analysis of how change efforts *failed* in organizations. Having studied dozens of transformation efforts across 15 years, he wrote an article called "Leading Change: Why Transformation Efforts Fail" in which he identified that change efforts fail for one of eight reasons, which are the converse of these eight steps for effective change. Your role in leading change is to identify any obstacles to transformation and uproot them rather than letting them undermine the success of your effort.

Step 6: Create Short-Term Wins

Success begets success.

One important conclusion from the law of diffusion of innovation is that Salesforce's Trailhead motto is not strictly true. We're *not* all trailblazers, at least not in every aspect of our lives. Most people, most of the time, are trail **followers**. And there's nothing wrong with that.

When it comes to implementing Salesforce DX or other DevOps initiatives, you can expect at most 2.5% of the population to be true trailblazers. To extend the analogy, these are the innovators who tromp through the wilderness and blaze (mark) trees to show others where to go. They're followed by the equally intrepid 13.5% of early adopters who by analogy are perhaps clearing the trail and ensuring that it's increasingly easy to follow.

What these trailblazers offer are examples of successes, small and large. And it is these successes that you need to amplify, advertise, and celebrate. Keep track of these successes and share them.

My brother is an extremely seasoned mountaineer, who has traversed almost every terrain you can imagine. But for myself and the majority of the population, the best we'll do is hiking well-marked trails, while staying in range of cellphone towers that allow us to double-check Google Maps. It's the same with most of the development teams at our organization. Their adoption will come when they see clear examples of other teams that have succeeded.

You can strategically choose small early wins to build everyone's confidence in this process. Nowhere is this more important than with refactoring your codebase to make use of Salesforce DX unlocked packages. Many people have tried and failed to convert their entire org into a single massive package. But the most effective approach is to first build a single relatively simple package and ensure you can deploy it across all of your orgs. Unlocked packages make it easy to add unpackaged metadata into a package, so once teams have established initial success with that process, they can build on that strength as their packages grow and multiply.

Create short-term targets, especially quick wins that you can confidently achieve. This gives everyone on the team increased confidence and helps pacify critics and those who might oppose your efforts.

Step 7: Build on the Change

Kotter argues that many change projects fail because victory is declared too early. Quick wins are only the beginning of what needs to be done to achieve long-term change, and your goal here is to effect a deep and lasting change in approach.

On the basis of your initial successes, keep building. Continue to set targets for training and adoption. And importantly, encourage teams to track their own development and delivery metrics and practice continuous improvement.

Consider establishing a center of excellence around DevOps or Salesforce DX practices. This provides the opportunity for disparate teams to share their challenges and successes. This kind of ongoing knowledge transfer is important.

Step 8: Anchor the Changes in Corporate Culture

The final stage is to ensure that these practices become embedded in corporate culture, so that they can become self-sustaining and are simply the way the company operates.

When I joined Appirio, the organization had an extremely well-established performance-oriented culture. From day one the executive team championed open communication, transparency, collaboration, and efficiency. In addition, the agile practices of working in sprints, using user stories, and so forth are second nature. There is still work to do to embed DevOps practices into the culture and to ensure they are as natural to the organization in the future as sprints and user stories are today.

It's important to see the broader context in which these practices exist, so that even sales and business people recognize that a reason our organizations can be so effective is that we empower our developers and continuously optimize and automate our software delivery processes. It is only when such practices become part of your corporate DNA that you can have confidence that this change effort will outlive any of the original instigators, and be passed from generation to generation of the organization, despite the constant churn and turnover many IT organizations face.

Summary

DevOps is a rich and growing area in the IT world. The foundations of DevOps have their roots in early automation that developers enacted to facilitate their own workflow. But the practices continue to mature, become clarified, and grow in adoption.

DevOps combines the management and cultural practices of lean software development with the many technical practices that enable continuous delivery. Its business impacts have been analyzed extensively through the *State of DevOps Reports* and other studies, which conclude that these practices bring benefits that reach far beyond the development team.

Because software development and delivery is increasingly central to achieving organizational missions, the significance of DevOps is growing as well. Organizations that implement the various capabilities that lead to high software delivery performance are twice as likely to meet or exceed their commercial and noncommercial goals.

The business benefits of using SaaS systems like Salesforce are very well established. But DevOps practices are not yet common in the Salesforce world. There is an enormous amount that Salesforce practitioners can learn from the experience and successes achieved elsewhere in the DevOps world. Salesforce DX unlocks the door to combining the benefits of SaaS with the benefits of DevOps and allowing our development teams to be as effective as possible.

PART II

Salesforce Dev

To illustrate the DevOps lifecycle, this part of the book summarizes how to build applications on the Salesforce platform, while Part 4: Salesforce Ops (Administration) summarizes administering Salesforce in production. Building things right and running them wisely are the two most important aspects of the development lifecycle, but in this book we touch on them only briefly. The vast majority of the book, Part 3: Innovation Delivery, covers the stages of deployment and testing that unite development and operations. Less has been written about that critical topic, but there are enormous efficiencies to be gained from doing it right.

Developing on the Salesforce platform is generally fun and straightforward, and there is an amazing array of learning resources available to help. Salesforce's developer documentation is excellent, as are Trailhead and the thousands of presentations given at Salesforce conferences and events every year. In addition, there is a wealth of books, blogs, tweets, and StackExchange posts you can find to educate you or help you untangle complex problems.

Although this section is brief, we introduce key concepts in development and architecture along with recommendations for where you can learn more.

CHAPTER 4

Developing on Salesforce

Developing on Salesforce means configuring the platform to meet the needs of your organization. It can take the form of clicks or code, but is frequently a combination of the two. The scope can range from tiny changes to sophisticated applications.

Here, we'll briefly introduce the Salesforce DX development lifecycle. We'll then look at development tools, click-based development, and code-based development. In the next chapter, we'll look at application architecture to introduce important principles to help you structure your development in a flexible and scalable way.

The Salesforce DX Dev Lifecycle

The basic elements of the Salesforce DX development lifecycle are a Salesforce development org, an IDE or code editor, version control, and a CI tool to perform automated tasks like deployments. Version control and CI tools are discussed at length in Chapter 7: The Delivery Pipeline, and the types and purposes of different Salesforce orgs are discussed in Chapter 6: Environment Management.

Salesforce has enabled sandbox development environments for many years. With DX, there are now two additional ways to create development environments: scratch orgs and cloned sandboxes. Scratch orgs are a flagship feature of Salesforce DX. They are short-lived orgs you create "from scratch" based on code and configuration stored in version control. Cloned sandboxes can allow developers to clone an integration sandbox that has work still under development, instead of only being able to clone the production org. The Salesforce CLI now makes it possible to clone a sandbox and automatically log in from the command line.

Changes made in that development environment need to be synced to version control so that automated processes can test and deploy those changes. One of the most helpful features of scratch orgs is the ability to perform a simple `source:push` and `source:pull` command to synchronize metadata between version control and your org.

A. Davis, *Mastering Salesforce DevOps*, https://doi.org/10.1007/978-1-4842-5473-8_4

That capability will soon also be available when developing on sandboxes. Changes can also be retrieved from sandboxes using either `source:retrieve` or `mdapi:retrieve`, depending on whether you are storing metadata in the "Source" format or the original "Metadata API" format mentioned in Chapter 2: Salesforce.

Salesforce developers typically have to modify multiple aspects of Salesforce to create a solution. For example, a typical feature might involve changes to a Lightning Web Component, an Apex class, a custom object, a Lightning record page, and a permission set. To deploy that feature to another environment, it's necessary to isolate all of the related metadata changes. But it's entirely possible to make changes in the Salesforce Setup UI without being sure what type(s) of metadata you're modifying.

If you're the lone developer in an org, working on a single feature, it's relatively easy to distinguish the metadata for that feature, especially if you're using version control. Just retrieve all the metadata and check for changes in that org since you began developing; whatever has changed must pertain to that feature. When dealing with small volumes of changes, small teams, and a moderate pace of innovation, it's not hard to manage Salesforce changes and deploy them across environments. But as teams begin to scale up, the pre-DX workflow begins to suffer from many limitations.

As soon as you put multiple developers in a single development org, it becomes much harder to isolate and deploy their changes independently. But when developers are working on separate orgs, integration is delayed and it becomes much harder for them to build on the work of others. Scratch orgs address this need by making it trivial to create a fresh new Salesforce environment that is up to speed with the metadata stored in version control. As you update your code repository, you can push those changes to your org, and as you make changes in the org, you can pull them down into version control without having to enumerate each type of metadata. Thus, if possible, you should develop in scratch orgs.

Unfortunately, it's still not practical for every team to use scratch orgs, and so you may find you need to keep using sandboxes for development. One very promising workflow is to automate the cloning of developer sandboxes from a single integration sandbox and then deploy changes as they're ready from those developer sandboxes into that integration org using a CI process. Recent updates to the Salesforce CLI make it possible to automate sandbox cloning and login.

One way or another, your job as a developer is to make changes in your development org, to commit them correctly to version control, and then to monitor automated tests and deployments to ensure your feature can be deployed without error. Setting up such

automation is the topic of later chapters, but once it's set up, developers can adopt this rhythm: build, commit, monitor.

Since version control is critical to this process, before beginning development, you must establish a code repository for your team. If a project has already been created, you'll need to clone that project to your local machine before proceeding.

There are two Salesforce DX development models: the org development model and the package development model. The org development model is the default approach, but the two are complementary. Your team may gradually migrate most of your metadata into packages, but there will always remain the need to manage some org-level metadata through the org development model. You can find a nice introduction to these different models on Trailhead at https://trailhead.salesforce.com/content/learn/modules/application-lifecycle-and-development-models.

The options available for developing on Salesforce are in flux, and most Salesforce developers are still getting up to speed on foundational concepts such as version control. This means that there's more variety in the development workflow than there was just a few years ago, and the optimal workflow of the future is not yet clear. This book attempts to present a comprehensive picture of the options, but expect things to evolve over the coming years.

If you are using the org development model, you will probably have a single repository representing all aspects of the org's metadata that are managed by the development team. Chapter 7: The Delivery Pipeline provides recommendations on an appropriate branching structure for this model.

If you are using the package development model, you will probably have one repository for org-level metadata and one or more repositories for managing package metadata. Dividing your code across multiple repositories makes it easier to set up automation for each repository, but can easily get confusing as to which repository contains which metadata.

The package development workflow lends itself to a simpler Git branching structure than the org development workflow. Until you develop tooling that can dynamically determine which packages have changed and should have new versions published, separate packages should be developed in separate code repositories.

The package development model implies that you are using scratch orgs. If you are using scratch orgs, you will need to periodically recreate these scratch orgs to ensure that they are clean and reflect the latest version of the codebase. If you are working with a complex set of metadata and package dependencies, you may find the scratch

org creation process takes a long time (up to an hour or more), so you may want to periodically precreate scratch orgs for upcoming work. The forthcoming Scratch Org Snapshots capability allows you to perform scratch org creation in advance and take a Snapshot of that org. The Snapshot can then be used to quickly create new orgs that are fully provisioned.

For those accustomed to working in long-lived sandboxes, it can feel frustrating to have to periodically recreate an entire development org. The purpose of recreating scratch orgs is to ensure that the entire application is fully represented in version control. This allows others on the team to create identical environments, and knowing all your dependencies limits the chances of confusion about why your applications don't work properly in testing and production environments.

Subtle variations between the orgs used for development and testing are a massive source of risk, confusion, and inefficiency. This is a hidden challenge that often goes unnoticed, and the time spent debugging these variations between orgs is generally not accounted for. But every moment spent debugging out-of-sync orgs is waste. Though it may seem like a radical solution, the regular destruction and recreation of scratch orgs is key to ensuring an efficient overall workflow.

You should commit changes to the code repository after each significant change. If you are making changes on a feature branch in Git, all of those changes will be associated with the name of that branch, and so your branch naming strategy can be a way of linking a work ticket number to a group of many changes.

When your development is complete and you're ready for review, you will merge it into your shared master branch (or create a merge request if your team requires that). Developing in scratch orgs unlocks the possibility of creating "Review Apps" so that other members of the team can review the developer's work in a live scratch org.

Review Apps are environments created dynamically from version control that provide an isolated environment to review features that may still be under development. The concept was popularized by Heroku. Review Apps should be used for most QA and testing activities, but they only contain the test data that you deploy and are generally not connected to external systems, so some testing might need to be done in an SIT org instead.

Many of the topics presented in this brief overview are covered in more detail in later chapters.

Development Tools

Some aspects of Salesforce development are done directly inside the Salesforce Setup UI, while other aspects are best handled with developer-specific tools. The critical step of externalizing changes in Salesforce to a version control system is always done outside of Salesforce. This section gives a brief introduction to key Salesforce development tools and how they are used.

The Salesforce CLI

The Salesforce CLI can be downloaded from `https://developer.salesforce.com/tools/sfdxcli`. This command-line interface is an essential part of your DX toolkit. Many Salesforce CLI commands will only run inside a Salesforce DX project. A Salesforce DX project is one which contains a file called `sfdx-project.json`. That file is used to store information about the folders and packages in your project.

You can quickly scaffold a new project by using the `sfdx project:create` command, or you can clone one of the sample repositories from the Salesforce DX Developer Guide to be able to use a CI system like CircleCI. All of these project templates contain `sfdx-project.json` along with other helpful files.

You can get help for any CLI command by adding the `-h` or `--help` parameter. You can also find detailed help in the Salesforce CLI Command Reference. The CLI commands also allow you to export their results in JSON format by adding the `--json` parameter. This capability unlocks the possibility of automating complex processes by extracting the results from one command and passing them as parameters to other commands. See the section on "Command-Line Scripts" in Chapter 9: Deploying for advice.

What's an Integrated Development Environment (IDE)?

An IDE is a code editor which brings together all the tools that developers need to write, track, debug, test, and deploy code. Some IDEs are built for a particular language (e.g., PyCharm is specific to Python), whereas others (like Eclipse and IntelliJ) support multiple languages. An IDE provides a code editor, but also offers things like syntax highlighting, code completion, debugging, and more. Most IDEs are extensible, allowing you to install plugins which add new functionality.

The Developer Console

The Dev Console is a web-based IDE that is built into Salesforce. It's accessible from the gear icon in the upper right of each Salesforce org. It provides a convenient way to edit Apex, Visualforce, or Lightning Components. It also allows you to review debug logs, set checkpoints, and run anonymous Apex or SOQL queries among other capabilities.

It does not provide you a way to track metadata in version control, do deployments, or work with Lightning Web Components, and it's no longer under active development. For certain tasks, like using logs to analyze performance, optimizing queries using the query plan, or checking Visualforce ViewState, the Developer Console is still my go-to tool, so it's worth becoming familiar with its capabilities, although it will become less relevant over time.

Workbench

Workbench is an unofficial Salesforce tool hosted at `https://workbench.developerforce.com`. It provides a user interface for numerous developer-focused tools. In particular, it provides a simple way to navigate Salesforce's various APIs and to test out API commands. Workbench exposes the Metadata API's ability to deploy or retrieve metadata and to describe metadata such as custom objects. It exposes the bulk API's ability to retrieve or modify data. And it allows you to execute SOQL or SOSL queries and anonymous Apex.

Workbench may never become part of your daily workflow, but it's important to know it exists, as it's the right tool for a wide variety of jobs.

The Forthcoming Web IDE

If this were a Salesforce conference, I would insert a forward-looking statement slide at this point. The Salesforce DX team is working on a web-based IDE that will encapsulate the capabilities of the Developer Console and Workbench (and thus allow those tools to be retired). At the same time, the Web IDE will allow teams to store complete sets of project files, run Salesforce CLI and Git commands, and interact with both Salesforce orgs and code repositories.

The initial goal is to get feature parity with the Developer Console and Workbench. Eventually this will provide a convenient IDE for teams who aren't able or don't wish to use desktop development tools.

Visual Studio Code

Salesforce built the original Force.com IDE on Eclipse, an open source IDE popular among Java developers. In 2018, Salesforce retired that IDE and began building a set of extensions on top of Visual Studio Code.

Visual Studio Code (VS Code) is a rare success story among developer tools. It's an open source code editor that first appeared in 2015. Within 3 short years, it became the world's most popular development environment.[1]

VS Code is the product of Microsoft, a company whose success and wealth have been built around proprietary commercial tools. Microsoft has not historically been well loved by the open source community, but VS Code is one of many such contributions[2] made under Satya Nadella's guidance. VS Code may even be reducing the number of people willing to pay for Microsoft's commercial code editors.

Whatever the history, Microsoft has built an editor that has won the hearts and minds of millions of developers. And by making the tool open source, they are benefitting from an amazing stream of innovation from hundreds of citizen developers who have contributed features and bug fixes. Not to mention the thousands of companies like Salesforce who have built extensions for VS Code.

So, what is VS Code? Why has it become so popular? And why, in particular, is it the new chosen platform for Salesforce to build their IDE on?

Visual Studio Code is a free, open source, cross-platform, multilingual IDE. VS Code has the speed and simplicity of Sublime Text, with the power and tool set of Eclipse or paid IDEs like IntelliJ. It's fast, simple, and clean; there's a fast-growing set of extensions for it; and it's adding new features every month.

Because it's popular, it's attracting innovation. In its short lifetime, VS Code has become the world's most popular IDE, with more than 50% of developers worldwide adopting it. Salesforce has deprecated the Force.com IDE and is solely focused on building tools for VS Code.

- Visual Studio Code is fast and can be installed on Windows, Mac, or Linux machines.
- It has support/extensions for 72+ programming languages out of the box.

[1] https://insights.stackoverflow.com/survey/2018/#development-environments-and-tools
[2] www.zdnet.com/article/microsoft-open-sources-its-entire-patent-portfolio/

- It gives built-in syntax highlighting and bracket matching in your code, as well as easy code navigation.
- It has a built-in terminal and a stepwise debugger that supports many languages, including the new Apex Debugger and Apex Replay Debugger.
- It supports Git natively so you can view diffs, commit, sync, and perform all the essential source control commands without leaving the editor.
- It has a wise set of defaults but is extensively configurable.

There are many blogs that have done "bake offs" or detailed feature comparisons between other popular editors (like Vim, Eclipse, Sublime Text, Atom, and TextMate). We won't repeat all of the details here, but instead just summarize some of the main comparisons as follows:

- It's more user-friendly than old-school editors like Vim, Emacs, Nano, Pico, and so on (anyone remember Edlin?).
- It's faster and more flexible than Eclipse.
- It has more robust built-in developer tools (like version control and debugging) compared to Sublime or Atom.
- It's also faster and handles large files better than Atom.
- It has far richer capabilities (like syntax highlighting and autoformatting) compared to TextMate, BBedit, or Notepad.
- It's free-er than IntelliJ or (the original, commercial) Visual Studio!

While Salesforce itself lives "in the cloud," professional developers tend to write code for Salesforce using desktop tools. For a long time, the Force.com IDE based on Eclipse was the standard Salesforce development environment. With the preview release of Salesforce DX in 2016, Salesforce launched a Force.com IDE 2.0, also based on Eclipse. But before long, they changed course, deprecated the Force.com IDE and are now exclusively promoting VS Code as their official editor.

One reason for the change of heart is the architecture that VS Code is built on. VS Code is built using Electron, a framework that allows you to build Windows, Mac, and Linux desktop applications using HTML, CSS, and JavaScript. This means that

to improve the user interface or add automation to VS Code, you can use the same technologies used to build web sites—the most common IT skillset in the world.[3] VS Code also uses the innovative concept of Language Servers—which allowed Salesforce to build a generic language server for Apex and Lightning[4] that can in theory be ported to Atom, Eclipse, or any other IDE that supports Language Server Protocol (LSP).

In the meantime, Salesforce was aware that MavensMate (especially in conjunction with Sublime Text) had become the open source editor of choice for many Salesforce developers who preferred its speed and simplicity to the older, rigid structure enforced by Eclipse. Sublime Text's simplicity was a key inspiration for VS Code's clean UI.

So by Dreamforce 2017, Salesforce had officially retired the Force.com IDE (including the newer IDE 2.0), in favor of VS Code.

In the meantime, Salesforce has continued to roll out innovation on top of VS code, and the VS Code team themselves have been releasing new features at a phenomenal pace.

VS Code should be your default choice for a Salesforce IDE. You can use it on your existing Salesforce orgs using the new metadata deploy and retrieve commands. This gives you a chance to get used to tools like the Apex Replay Debugger (a superior way to handle debug logs), test management tool, and more.

Despite these developments, VS Code still has frustrating gaps, especially for those developing on sandboxes. As of this writing, it does not provide any warning if you are pushing updates to a sandbox that will overwrite others' work. Those limitations will eventually go away, but you should also seriously consider one of the IDEs mentioned below in the meantime.

Other Salesforce IDEs

There are currently two commercial IDEs for Salesforce that provide extremely robust capabilities: The Welkin Suite and Illuminated Cloud. The Welkin Suite is based on Microsoft's Visual Studio (the commercial one, not VS Code) but is downloaded as a standalone application. Illuminated Cloud is a plugin for the IntelliJ and WebStorm IDEs by JetBrains.

---

[3] https://insights.stackoverflow.com/survey/2018/#technology-programming-scripting-and-markup-languages

[4] https://youtu.be/eBOVoYOb2V8?t=861

Both Visual Studio and IntelliJ are exceptional development environments and are loved by millions of developers. They both now offer free tiers, but their popularity has been eclipsed as VS Code's has exploded.

Illuminated Cloud and The Welkin Suite fill major gaps in the Salesforce extensions for VS Code. They both charge an annual fee, but will quickly pay for themselves in terms of time and agony saved for each developer. The Welkin Suite is somewhat more expensive, but is supported by a larger team of developers. Illuminated Cloud is the work of Scott Wells, who has supported it tirelessly and has an amazing knowledge of the platform and the challenges faced by developers.

Both of these tools innovate extensively and continue to have promising futures, even as Salesforce evolves their free alternative. The Welkin Suite created a replay debugger several years before this was available in VS Code. And Illuminated Cloud combines IntelliJ's excellent features such as code completion and task management integration with support for all Salesforce languages, metadata types, and development models.

In addition to these, it's worth mentioning three other tools. ForceCode[5] was one of the earliest VS Code extensions for Salesforce. It was created by John Nelson while working for CodeScience. The project had been deprecated after Salesforce released their extensions for VS Code, but I learned recently that it has come back to life. Among other good qualities, it helps compare your local copy of code with the latest version in your Salesforce org and includes tools to manage the process of building complex single-page apps using JavaScript frameworks and deploying them to the Salesforce platform.

MavensMate is now retired but was largely responsible for convincing the Salesforce developer ecosystem that there were faster and simpler alternatives to the Force.com IDE. Joe Ferraro labored for years to provide MavensMate, doing an enormous service to the Salesforce developer community.

Aside.io is a web-based IDE that was also very popular with my colleagues. It has the benefit of not requiring any local software installation. But as a result, it does not provide a method to interface with version control. Aside.io may be discontinued soon, but its founder has committed to open sourcing the project if possible.

[5] https://marketplace.visualstudio.com/items?itemName=JohnAaronNelson

Metadata (Config and Code)

For those new to Salesforce development, it's useful to reiterate that Salesforce does not allow for local development because changes are necessarily compiled and run on a Salesforce instance and cannot be run locally. Lightning Web Components are some exception to that, and LWC local development capability is now available. The JavaScript used in static resources is also available for local development, but these are typically only small parts of a Salesforce application.

What is possible, however, is to download your Salesforce configuration as metadata that can be stored, tracked, updated, and deployed back to a Salesforce instance.

What Is Metadata?

Most Salesforce customizations are represented as metadata components. These metadata components are files that can be retrieved from the server (Salesforce instance) and modified locally, then saved back to the server or deployed to another environment. A detailed list of available metadata items can be found in the Metadata API documentation.[6]

These metadata files are largely stored as XML, although some are code, and some new types are JSON. It's this metadata representation that can be stored in version control and is the basis for automated processes such as static analysis and automated deployments.

Metadata API File Format and `package.xml`

As explained in Chapter 9: Deploying, the key technology that enables retrieving and deploying metadata from a Salesforce org is the Metadata API. The Metadata API uses a file called `package.xml` to enumerate the metadata that should be deployed or retrieved. Some developer tools like MavensMate rely on this file as a representation of the metadata in your org.

The use of `package.xml` can lead to confusing results if there's a mismatch between its contents and the metadata files you're working with. So an increasing number of tools are doing away with this file and instead automatically generating it as needed.

---

[6] https://developer.salesforce.com/docs/atlas.en-us.api_meta.meta/api_meta/meta_types_list.htm

For example, the Salesforce DX source format does not include this file at all. Nevertheless, it's dynamically generated behind the scenes when communicating with the Metadata API.

The Metadata API explicitly deals with files using an `src/` directory that contains the `package.xml` file itself as well as the individual metadata for the project. These files are organized into folders based on the type of metadata as shown in Listing 4-1. Each metadata item is its own file within that folder.

Listing 4-1. The native Metadata API file and folder structure

```
src
├── applications
│   └── DreamHouse.app
├── layouts
│   ├── Broker__c-Broker Layout.layout
│   └── Property__c-Property Layout.layout
├── classes
│   ├── ClassA.cls
│   ├── ClassA.cls-meta.xml
│   ├── ClassA_Test.cls
│   └── ClassA_Test.cls-meta.xml
├── pages
│   ├── PageA.page
│   └── PageA.page-meta.xml
└── package.xml
```

Note that some metadata types like `applications` and `layouts` have a single metadata file to represent a single item, while other types like `classes` and `pages` use two or more files. Files like `ClassA.cls` contain the actual metadata body (in this case an Apex class), while files like `ClassA.cls-meta.xml` are called "sidecar files" and store some accompanying metadata. Sidecar files are typically very small XML files.

One of the innovations of the Salesforce DX source format is the ability to group related metadata into subfolders that can eventually be published as packages. That is not possible in the native Metadata API format and is a key benefit of moving to Salesforce DX.

Converting Between Salesforce DX and Metadata API Source Formats

There are special commands to convert between the Salesforce DX metadata format and the Metadata API format. The Salesforce DX Developer Guide and some Trailhead modules[7] and video tutorials[8] describe this process in more detail. Here we provide just a brief overview.

`sfdx force:project:create` allows you to create a new set of project files in the Salesforce DX format. But it's often the case that you want to convert an existing set of metadata files into Salesforce DX "source format."

`sfdx force:mdapi:convert` operates on projects stored in the native Metadata API format and converts them to the "source format." All files in the `src` directory are converted into files in the default folder specified in `sfdx-project.json` (typically `force-app/main/default/`). Large `.object` files are decomposed into smaller components, zipped static resources are decompressed, and so forth. Initially these files are not grouped into subdirectories, but after being converted, you can create subdirectories to group metadata into packages.

`sfdx force:source:convert` performs the opposite conversion, taking metadata files in the SFDX source format and converting them into the native Metadata API format. Even if you have grouped your source format metadata into many subfolders, once converted into the Metadata API format, they will all be in a single `src/` folder. This process will autogenerate the `package.xml` file mentioned earlier. Note that this is a "lossy" conversion; if you convert these files back to the DX source format, you will lose the folder structure.

What Metadata Should You Not Track?

The main goal of this book is to provide advice on building a CI/CD pipeline for Salesforce. That pipeline becomes a delivery vehicle for metadata to allow it to be built in a development environment, promoted for testing, and finally delivered to production.

[7] https://trailhead.salesforce.com/en/modules/sfdx_app_dev/units/sfdx_app_dev_deploy

[8] www.youtube.com/watch?v=6lNG6iFVGQg&list=PLp3OcEU4IpnBX2yZWJw7jjMXvsFIltM57&index=2

There are however types of metadata such as those shown in Table 4-1 that generally should not be included in the code repository and CI/CD process. There can be temporary and long-term exceptions to this rule depending on the needs of the project team. Think of the things that you include in the CI/CD process as being "controlled by developers and admins," as opposed to being "controlled by end users." The essence of continuous delivery is being able to reliably recreate development, testing, and production environments and code. And for this reason all core functionality (custom objects and fields, business logic, even page layouts) should be controlled by this development process. But functionality like reports and dashboards are a great example of metadata that can safely be controlled by business users, since they may need to change frequently and with rare exceptions changing them will not cause side effects elsewhere in the system.

If you exclude these items from your code repository, you should also add them to your `.forceignore` file to prevent them from sneaking back in.

Table 4-1. *Types of metadata that might be excluded from CI/CD*

Metadata Type	Reason to Exclude from CI/CD
Certificate	Certificates are generally environment-specific and should be kept secure.
ConnectedApp	Connected Apps are generally environment-specific. If you automate their deployment, you will need to dynamically replace some parameters.
Dashboard	These are often changed frequently by users in production and should not go through the development lifecycle unless they are a dependency for your code or metadata.
Document	These are often changed frequently by users in production and should not go through the development lifecycle unless they are a dependency for your code or metadata.
EmailTemplate	These are often changed frequently by users in production and should not go through the development lifecycle unless they are a dependency for your code or metadata. An exception is VisualForce email templates or other templates that use embedded code and may require a careful development process.

(*continued*)

Table 4-1. *(continued)*

Metadata Type	Reason to Exclude from CI/CD
InstalledPackage	You can't control the installation order. Use `sfdx package` commands to install packages instead.
Layout	When following the package development model, page layouts can be problematic. They should be managed at the org level for any objects that straddle multiple packages.
NamedCredential	Named Credentials are generally environment-specific. If you automate their deployment, you will need to dynamically replace some parameters.
PlatformCachePartition	These are typically environment-specific.
Profile	Profiles are the fussiest of all metadata types, as explained below. Without special automation, you will find it easier to manage these manually.
Report	These are often changed frequently by users in production and should not go through the development lifecycle unless they are a dependency for your code or metadata.
SamlSsoConfig	SAML SSO Configuration is usually environment-specific. If you automate their deployment, you will need to dynamically replace some parameters.
Site	The Site metadata type represents Communities, but is stored as a binary file and can change unpredictably, making it a poor candidate for CI/CD.

Retrieving Changes

After modifying items in a development org, you need to retrieve those changes and merge them into your local working directory so that you can track changes and deploy them using CI/CD.

The Salesforce CLI and the different IDEs mentioned earlier all provide mechanisms to retrieve metadata. The commercial Salesforce release management tools introduced in Chapter 9: Deploying also provide mechanisms for doing this.

The way the Metadata API behaves can make retrieving metadata very awkward without the help of a specialized tool. This is especially true when working in the native Metadata API format. A single `.object` file can contain hundreds of child metadata items. If you request to retrieve one of those child items, such as a field, your request will overwrite the entire `.object` file with just the metadata for that one item.

Profiles are famously awkward to deal with. A profile can contain thousands of permission details such as field-level security. But when you issue a simple retrieve command for a profile, it will retrieve only the small subset called "user permissions." To retrieve the field-level security part of a profile, you have to retrieve both the profile and the appropriate field. As mentioned earlier, this will overwrite the local metadata file for the object and will also overwrite any other permissions in the profile file. Even more confusingly, to retrieve the page layout assignment for a particular record type, you have to retrieve the profile, the page layout, and the particular record type. I could go on.

Why such bizarre and unfortunate behavior? It's because the Metadata API was not originally designed to create a single representation of the org that was suitable to be stored in version control. This is one of the key reasons why it's important to use tools that are specialized for Salesforce development, and one of the key reasons why naively attempting to build a Salesforce CI/CD process without specialized tools will end in tears.

Adopting a specialized Salesforce IDE or release management tool will pay for itself very quickly (especially if you use the free ones!). These tools have solved these problems and allow you to follow a rational workflow for retrieving, tracking, and deploying metadata.

The Salesforce DX source synchronization process for scratch orgs also addresses these challenges. The SFDX source format is designed to store metadata for use in version control and automated deployments. And the source synchronization capability of scratch orgs handles complex retrieves and merges automagically.

Making Changes

Changes to a Salesforce org can be made either using the Salesforce UI or by modifying the metadata in your local IDE and compiling it to the Salesforce org.

Most configuration changes should be made with the UI to ensure that the metadata remains coherent, while most code changes are made from the IDE. The Salesforce-specific IDEs perform a two-step process when saving. They first save your file locally and then attempt to deploy them to your development org. If the deployment is unsuccessful, you will receive an error that prompts you to fix your change. In the case of code files, deploying your metadata files will also compile your code and show you any compile-time errors. If the deployment is successful, your update will be saved to the org.

Note that you have to remain attentive to any errors returned by your IDE. It's entirely possible to commit local metadata files to version control in an invalid format, which will cause downstream failures if you attempt to deploy them. It's the developer's responsibility to ensure that they can make a successful round-trip from their local metadata file to the Salesforce org and back.

Salesforce DX scratch orgs and source synchronization address these challenges. Scratch orgs are generally designed for a single user or for limited collaboration between one dev, one admin, and maybe people testing or demoing new features. This bypasses the challenge of possibly overwriting others' work in a shared sandbox.

The source synchronization process also makes deployments to scratch orgs very fast and simple. Local metadata is first analyzed to see if it has changed (using a timestamp and a hash), and only metadata that has been changed is pushed, which makes the deployments much faster. There's no need to specify which specific files should be pushed to the scratch org; every changed file will be pushed using a single command.

Manual Changes

Almost all of the steps required for a deployment can be automated using metadata and the Salesforce CLI, although it takes time to get familiar with the way that UI changes are represented in metadata. There are however some changes that cannot be tracked or deployed in this way. When manual configuration is needed for functionality to work, developers or admins should track and document the required steps in whatever project tracking tool is being used. The instructions should be as detailed as necessary for whoever will eventually perform those manual steps.

It's at this point that the dreams of fully automated deployments bump up against a more manual reality. As mentioned in Chapter 9: Deploying, many of the things that people think are manual steps can in fact be automated. And the Salesforce Metadata API teams keep working on addressing gaps in the metadata.

Traditionally, when developing on sandboxes, developers need to have extensive knowledge of many different metadata types and do research to determine if and how that metadata can be downloaded. The source synchronization process available in scratch orgs and soon also in sandboxes addresses this problem by automatically identifying what was changed through the Setup UI and downloading the appropriate metadata. The productivity gains from this cannot be overstated.

Click-Based Development on Salesforce

There are roughly 250 different types of Salesforce metadata that are configured declaratively, using "clicks not code," on the Salesforce platform. This configuration can be as simple as toggling settings and as complex as using sophisticated "Builders" to create complex UIs, data models, and business logic. The App Builder, Schema Builder, Flow Builder, Process Builder, and Community Builder provide drag-and-drop components for these purposes, each with their own configurable settings.

The degree to which Salesforce can be configured in this way is a defining characteristic of the platform. It means that even code-based Salesforce developers need to be familiar with at least the basic declarative configuration tools. It also means that one can spend their entire career building and customizing Salesforce and never leave the Salesforce UI.

Development—No-Code, Low-Code, and Pro-Code

Salesforce sometimes distinguishes three types of developers—no-code, low-code, or pro-code—based on their skills and preferred tools. The term "developer" implies someone who creates something systematically and by phases. This is certainly true of those who code, since the process involves extensive testing and iteration. But it's equally true of those who use graphical tools. The initial iteration of a Flow might be quite simple. It then needs to be tested, edge cases need to be identified, logic can be added, and so forth.

It's worth remembering that code itself is a user interface—it's just a text-based UI rather than a graphical one. The history of computer science is largely the history of building increasingly high-level abstractions to make logic more clear for humans. Assembly language was created to free people from having to write numeric instructions in machine code. Higher-level languages like C freed people from having to deal with esoteric assembly language. Languages like JavaScript layer on further abstractions and simplifications, eliminate the process of compiling code, and give access to hundreds of thousands of prebuilt modules that encapsulate solutions to common problems.

Each transition to a higher level has made the language more accessible, at the cost of some performance. Those who are familiar with working at lower levels of abstraction can boast of building more optimized solutions. But a performance cost is often a very worthwhile sacrifice for a solution that is easier to understand and maintain over time.

And so in Salesforce, "pro-code" developers are those who are comfortable or prefer working directly with Apex, Visualforce, Lightning, or Salesforce APIs. "Low-code" developers are those who are more comfortable working with declarative tools, but who are nevertheless comfortable creating or maintaining small and strategic bits of code. And "no-code" developers are those who build and maintain customizations entirely using declarative tools.

"No-code" developers are extremely common in the Salesforce world. But the idea of developing without code is still a nascent movement in the IT world. Graphical abstractions for UI and data models are extremely common, for example, with tools that allow you to build web pages without dipping into HTML or build relational databases visually. Graphical abstractions of logic are less common; but even the most sophisticated programmers often crave and create visual representations of complex code like UML diagrams. In this respect, the future is already here on the Salesforce platform.

Salesforce Flows actually use Visualforce or Lightning to generate pages. In that sense, Flows are truly higher-order abstractions built over top Salesforce coding technology. As with other abstractions, Flows and Processes do not perform as well as pure Apex or Lightning solutions, and their metadata format is arcane. But they allow business processes to be visualized and edited by a far larger group of users, which is a massive boon for maintainability. They are one of many examples where Salesforce is democratizing development. While pure coders may be dismissive of solutions that trade efficiency for simplicity, history shows that this can be a massive strategic advantage.

But don't expect code to go away anytime soon. Salesforce themselves had to retreat from their original "clicks not code" motto. And "pro-code" development options like Web Components, Git, and CI/CD are becoming increasingly easy and popular on Salesforce. Just as I would have been hard-pressed to write this book using graphical tools, there's often no substitute for the freedom and flexibility of text-based code to express complex scenarios.

Declarative Development Tools

Entire books have been written about Salesforce declarative tools like Community Builder, so I'm making no attempt to cover that topic exhaustively here.

What matters for our purposes is that any declarative changes that are made in Salesforce need to be converted into a textual form if they are to be tracked in version

control and migrated to other environments using the tools described in Chapter 9: Deploying. This is the job of the Metadata API, described in detail in that chapter. The Metadata API provides a text-based representation of declarative configuration, mostly in XML, that can be inspected, tracked, and deployed to other environments.

In most cases, the purpose of capturing declarative configuration as text is simply to retrieve it from one Salesforce environment and deploy it to another environment. Along the way, there certainly is manual or automated processing that can be done on that metadata, some of which is described later in this book. But just as the declarative tools themselves range from simple to complex, so too does this metadata representation. I feel extremely comfortable viewing and updating the XML that describes custom objects, fields, and validation rules. But I don't know anyone who regularly edits the metadata representations of Flows and Processes.

There are countless tips and tricks that could be shared about working with declarative metadata; some of these I know, many of them I don't. So I'll constrain this discussion to a few points about a few of the most complex declarative Builders.

Lightning App Builder

The Lightning App Builder is a way to create UIs by assembling Lightning Components. From a metadata point of view, the final result of using the App Builder is a metadata item of type `Flexipage`. This is an XML representation of the page that includes the page layout, the components on the page, component properties, and some other metadata such as platform actions available for that page. This metadata is stored in a single file and is easy to version and deploy.

Community Builder

Community Builder is similar to Lightning App Builder in that it is used to create and configure user interfaces with drag-and-drop elements. But it adds considerable sophistication in that it can be used to create entire Communities—multipage web sites—and to define the overarching structure of the site as well as individual pages. Community Builder is a massive area of growth and focus for Salesforce, and they're working on a unified "Experience" model that can also encompass the web site builders associated with Marketing Cloud and Commerce Cloud.

One massive disadvantage of Community pages is that until recently they didn't have human-readable metadata contents. That's changing with the release of the

`ExperienceBundle` metadata type, although that's still in Developer Preview as of this writing. In marked contrast to most Salesforce metadata, ExperienceBundles are stored as JSON files. Salesforce began in 1998 and is mostly written in Java, the most promising programming language from that time. Salesforce also relies heavily on the data storage and exchange formats which were popular at that time, XML and SOAP, which are very well supported by Java. JSON is a newer and simpler standard that has grown in popularity alongside JavaScript, since it allows for smaller and more readable files, which can be parsed natively by JavaScript.

Although it's currently possible to deploy entire communities using the `Site`, `Network`, and `SiteDotCom` metadata types, the `Site` is stored as a binary file which changes each time it's retrieved, and so isn't suitable for version control or continuous delivery. As ExperienceBundles become widely adopted, it should become possible to selectively deploy aspects of a Community without redeploying the entire thing. Until then, teams must either deploy the entire Site or manually migrate Community configuration from one Salesforce org to another.

Process and Flow Builders

Process Builder is used to create business logic that is triggered by an event such as a record update. Processes are necessarily linear sequences of steps that are performed one after another, although there are an increasing number of available options.

Flow Builder is used to create more complex logic and can also be used to create user interfaces and to define data flow across screens. Processes are built on the same underlying technology as Flows, and they are both retrieved using the metadata type `Flow`.

For many releases, managing Flow metadata also required managing metadata for FlowDefinition. Each Flow or Process version was stored as a separate `Flow` file, and a corresponding `FlowDefinition` file specified which version of the flow was active. Since Winter '19, only the active Flow version is retrieved, FlowDefinitions are not needed, and Flows can be deployed as active under certain circumstances. Since Flow versions update the same filename, you can keep a clear history of changes in version control.

Flows and Processes are one of a small number of metadata types where Salesforce manages version numbers internally and allows versions to be updated or rolled back from the Setup UI.

This is a very powerful capability for many reasons. First, Flows are asynchronous and may be waiting for user input for many minutes. Flows store the user's state in a

`FlowInterview` object, along with which version of the Flow is being used. Even if a different version is activated while a user is still progressing through that Flow, they will be able to complete the process using the same Flow version they started with.

Second, this allows for versions of a Flow to be deployed without being immediately released. As explained in Chapter 10: Releasing to Users, the ability to deploy without releasing is a key capability that enables continuous delivery. While Flow versions do not allow you to release to only a subset of users, they certainly enable you to roll out or roll back a flow version on your own schedule, regardless of when they were deployed to an org.

Finally, Flow versions provide a form of declarative version control. The use of traditional version control is still far from ubiquitous on Salesforce. Declarative developers building Processes and Flows are grappling with complex logic and UIs that might take days or weeks to build. As with code-based development, it's invaluable to be able to take a snapshot of your work and begin experimenting with a new approach, confident that you can always roll back to a previous version. By dispensing with `FlowDefinition` metadata, Salesforce is making it easier to manage Flows using traditional version control like Git while still preserving the declarative versioning of Flows.

There's one final sense in which Flows and Processes are unique among declarative metadata. Salesforce has begun to enforce code coverage requirements on these, just as they do with Apex code. As we'll discuss in Chapter 8: Quality and Testing, Apex tests are not limited to only validating Apex code. They can validate any form of backend business logic in Salesforce, including validation and workflow rules, Processes, and autolaunched Flows (those which don't require a UI).

The only case when code coverage is required for Processes and Flows is when they are deployed as Active. By default, they are deployed in a Draft status and have to be manually activated. You can enable the ability to deploy these as active from **Setup ➤ Process Automation Settings ➤ Deploy processes and flows as active**. If you choose this option, then you must ensure that you have Apex tests that exercise these flows. For example, if a Process is triggered any time you create or update an Opportunity with a particular record type, you must have an Apex test that performs such an action.

This is a fairly soft requirement, in that not every logical execution scenario has to be tested. So it doesn't require tests that are as detailed as those which test Apex classes or triggers, where 75% of all the *lines* have to be covered. But you have to at least touch those Flows and Processes.

Data Management

The very core of Salesforce is a relational database that can be easily managed using clicks not code. This is the earliest configurable part of the platform and is still the most heavily used. Admins can use graphical tools to define objects and fields, connect them to other database objects, place them on page layouts, and define who can see them.

A complete explanation of this topic is far outside the scope of this book, but it's worth mentioning a few points that have an impact on the overall flow of changes from development to production.

Managing the Schema

"Schema" is a fancy name for the data structures in a relational database. In the simplest terms, it defines what data you want to store and how that data is related to other data. In a traditional database, you define spreadsheet-like tables to hold each category of data, in which each "row" is a record, each "column" is a field, and some special columns can be "lookup relationships" to rows in other tables. In Salesforce, tables are referred to as "Objects" (like the Account Object) because the definition of that object includes far more than just the data structure.

Creating a new schema object or field is typically done using multistep wizards, and this is often the first configuration activity that Salesforce admins and devs will learn. For the impatient, Salesforce provides a Schema Builder that allows you to create objects and fields quickly while bypassing nonessential steps in the wizard.

Although this process is relatively quick and easy, it can have big impacts on the rest of the org. UI, reporting, and business logic are built around displaying, analyzing, and manipulating data, and so the schema is fundamental to most other customizations.

When people first begin to set up version control and continuous delivery for Salesforce, it's tempting to only track code. Code is certainly the most important candidate for version control, but code in Salesforce is largely built for retrieving and manipulating database objects, and so without tracking the schema you don't have a complete picture stored in version control. If you attempt to automatically deploy code to later environments, your deployments will fail unless the necessary objects and fields are already in place.

For these reasons, you should manage the schema in your testing and production environments using version control and a continuous delivery process. You should not allow any manual changes to the schema in those environments, unless you have a

high tolerance for confusion and inefficiency. You can and should modify objects and fields in a development environment and deploy from there along with the rest of your configuration.

As mentioned, there's a lot more to Salesforce objects than just a data table. This is reflected in the `object` metadata format, which includes fields, validation rules, compact layouts, and record types among other components. It's for this reason that critical objects in your data model like Accounts can easily have an object metadata file that grows to tens or hundreds of thousands of lines. Version control tools like Git excel at managing and merging line-by-line changes to code files. But the default `diff` tool included with Git gets extremely confused when comparing repetitive blocks of XML metadata, making such large files very hard for teams to manage and collaborate on.

The Salesforce DX source format "decomposes" each object file into a folder, with files for each child component organized into subfolders by metadata type. Breaking large XML metadata into many files makes it straightforward for teams to manage and deploy the schema collaboratively.

Changing the Schema

Most schema changes lend themselves well to being propagated from dev to test to production. There is, however, one caveat and two big exceptions. The caveat is that some seemingly innocent schema changes can cause data loss or other problems. The two exceptions where schema changes can't (or shouldn't) be propagated automatically are when changing the object/field names and when changing field types.

It is important to emphasize to your developers and admins that their changes can and will have an impact on production data and users, and that it is their responsibility to envision not only the end state that they want to build but also the impact of any changes to existing functionality. It's important to facilitate the free flow of innovation, but here are just a few examples where schema changes require a skeptical eye:

- Increasing the length of a field is usually safe, but decreasing its length can truncate data and cause data loss.
- Increasing the length of a field can cause problems if you're integrating with external systems and those systems truncate or refuse to accept longer data values.
- Adding validation rules is part of business logic and must be tested carefully since it can cause Apex tests and integrations to break.

- Disabling field or object history tracking will delete that historical data unless you're using Salesforce Shield.
- Making a field required can also break Apex tests and integrations and is not a deployable change unless that field is populated for every record in the target org.

Don't Change DeveloperNames

The two types of schema change that are truly problematic in a continuous delivery process are changing object/field names and changing field types. Every type of Salesforce metadata, including objects and fields, has a DeveloperName (aka API name) that uniquely identifies it. That name is usually created dynamically when you specify the label or display name for the metadata. For example, if I create a field labeled "Implementation Manager," the New Custom Field Wizard will suggest the field name `Implementation_Manager__c`. When you retrieve the metadata for that new field, the field name appears as the `<fullName>` property in the XML and the name of the field file if you're using the Salesforce DX source format.

Let's say you decide to change the label for that field to "Success Manager." It's tempting to also want to change the field name to `Success_Manager__c`. If you've already deployed this field to other environments, **DON'T CHANGE ITS NAME**. Take a cue from Salesforce, who regularly rebrands their features, but doesn't change their names on the backend. Einstein Analytics is still known as `Wave` in the metadata. Communities and Sites were rebranded a decade ago, but are still referenced as `Picasso` in some parts of the metadata. The list goes on and on.

The problem with changing developer names is that when you deploy the new name, it's created as an entirely new field, object, and so on. It's treated as something new and unknown; none of the data is carried over from the old object. Not only that, but you'll have to update any code, formulas, or external integrations that relied on it. **DON'T DO IT**, change the field *label* to update the UI, but leave the underlying field name the way it is. You can add an explanatory note in the field description if you think that will help.

Changing Field Types

What about changing field types? A groan is arising from the depths of my heart when I think about this activity; it's generally a big pain to propagate these changes. Here are some recommendations about addressing this.

First, the basics: every field in a Salesforce object has a field type. There are currently 23 possible field types including text, picklist, number, lookup relationship, and formula. Field types determine the storage size for fields and provide built-in data validation: you can't put text in a number field, lookup relationship fields have to point to another record, and so on. Twenty-three types of fields means there are theoretically $23 * 22 = 506$ permutations of field type changes. But there are limitations on this, so, for example, it's not possible to change a field from a formula into any other type or from any other type into a formula. Salesforce provides a help article outlining those restrictions and the cases in which data loss can occur, such as changing from a text field to a number field.[9] That article is an important reference when considering a field type change.

Why would you ever need or want to change the type of a field? There's always uncertainty in the development process. One common example is changing from a text field (which allows any value) to a picklist field (where only certain values are valid) or vice versa. The team may create a text field to store a particular value, but later realize that it's important to standardize the options to improve data quality and generate better reports.

The book *Refactoring Databases*[10] arose from the recognition that data models need to evolve over time and introduced the concept of "evolutionary database development" to address this challenging process. They provide numerous recommendations, not all of which are relevant for Salesforce, but some of which can be summarized here.

1. Delay creating (or at least deploying) objects and fields until you are actually ready to use them. It's very easy to modify your schema in your development org, but can be challenging to propagate changes, especially if you're already storing data.
2. Think long and hard about whether you actually need to change the type of a field. I've facilitated field type changes for developers, only to facilitate changing them back, and then changing them once again. Such indecision can strain your friendships.

[9] https://help.salesforce.com/articleView?id=notes_on_changing_custom_field_types.htm

[10] Scott W. Ambler and Pramodkumar J. Sadalage, *Refactoring Databases: Evolutionary Database Design* (Addison-Wesley Professional, 2006).

3. Experiment with the field type change in your development environment to see if it's even possible. Create a new field of the original type, and then change that field to see if it's permitted and what warnings are displayed. Delete that field when you're done.
4. Assess the risk of data loss from any field change, and get sign-off from the appropriate business stakeholders on the change.
5. Assess whether any external integrations connect to this field, to ensure they won't break or introduce bad data.
6. If you decide to proceed, you can change the field type in your development environment and attempt to deploy it. Some field type changes can be successfully propagated using the Metadata API. If your deployment succeeds, you're lucky.
7. If your deployment does not succeed, you have two options: manually change the field type or create a new field and migrate data.
8. You can manually change the field type in all other environments as a way of propagating this change. This is tedious once you become accustomed to continuous delivery, but may be your simplest option. The problem with this is if you have a long lead time to get changes into production, since it may require careful choreography to facilitate a large release and also to make manual changes such as field type changes, especially if those changes are required for work done weeks or months in the past. This is one of many arguments for doing small and frequent releases.
9. You can also create a new field of the new type and then migrate or synchronize data from the old field. This is necessary for certain field type changes, especially if the field is heavily referenced by code or other configuration. Migrating data from the old field to the new field can be done using a data management tool or by using batch or anonymous Apex, depending on how much data you need to move. Batch Apex can be used to iterate across thousands or millions of records. You query records which have the old field populated and the new field empty, and you copy

values from the old field to the new field, transforming them along the way. The Metadata API allows you to mark the old field as deprecated if you want and eventually to delete it.

10. In circumstances where you need to change a heavily used field in a production environment without incurring downtime, you can actually create a data synchronization between the old and new fields. This is the most elegant solution and could involve Apex, Processes, or Workflow rules to update the new field with the old value and the old field with the new value whenever they change. While complicated, this allows for true refactoring and enables you to deploy a complete process from development to production without requiring any manual steps. Needless to say, you should test the heck out of this process before deploying it. Once it's in place and your refactoring is complete, you can delete the old field.

Bulk Database Operations

In addition to field type changes, there may be other bulk database operations you need to perform. Many of my colleagues at Appirio specialized in transforming and migrating data from legacy systems into Salesforce, and for this there are specialized tools and skills you might need. The Salesforce data management applications Odaseva and OwnBackup both provide a wealth of tools that you might use to perform large-scale data operations. The free Salesforce Data Loader and MuleSoft's dataloader.io are excellent no-frills options to help with this process, especially in combination with clever use of spreadsheets for data transformations.

As mentioned earlier, the use of anonymous Apex and batch Apex are both excellent methods to perform bulk database operations, although you should approach this with due caution. Anonymous Apex allows you to build an Apex script so you can query, transform, and create/update/delete data as needed. Such scripts should generally be stored in version control (so you don't forget what you did), but are generally run through the developer console or the Salesforce CLI. Test such scripts very carefully. Anonymous Apex allows you to transform up to 10,000 records at a time (the DML limit). Combined with a query that is smart enough to know which records have not been processed, you can script or manually trigger iterations of this code until the job is done.

Batch Apex handles the iterations for you and is designed to process large volumes of data over time. Batch Apex is an Apex class that uses the `Database.Batchable` interface to define `start`, `execute`, and `finish` methods that iterate across your database 200 records at a time. This can be used either for one-time or ongoing processing as needed.

Data Backup

Unless you're unusually paranoid, you don't need to back up your data out of fear that Salesforce will lose it. They have excellent redundant backup systems that distribute the backup across multiple data centers. What you need to worry about is the actions of your users and yourself. Remember those bulk database operations I just mentioned? Remember how I said you should test them extremely well? I think you're getting the picture.

While unlikely, you could be subject to malicious users or hackers who intentionally delete data. More likely, unwitting users or admins might make sweeping changes that cause data loss or corruption. And it's also possible for developers to introduce batch Apex, processes, triggers, workflows, or integrations that cause sweeping and undesired changes.

For all of these reasons, data backup is important. Your free option is the monthly export service accessible in **Setup ➤ Data ➤ Data Export**. That will generate a one-time or scheduled downloadable backup. You should practice and ensure you have the skills on hand to restore some or all of that data in an emergency. Your concierge options are data management services such as AutoRABIT Vault, OwnBackup, or Odaseva who can provide data backup and recovery tools and services.

Configuration Data Management

One of the weakest links in the continuous delivery systems I have built and seen for Salesforce is the management of configuration data. Salesforce release managers and release management tools generally specialize in migrating configuration **metadata**. The Salesforce Metadata API provides a mechanism for retrieving and deploying this metadata. While there are some complexities in this process, Salesforce actually takes care of most of the complexity for you. When you do a deployment, Salesforce checks the validity of your code and configuration, ensures referential integrity and idempotency, assesses differences, sequences the metadata loads, and more. There's an enormous amount of intelligence being leveraged behind the scenes.

If you need to migrate complex configuration data, you will need to address those problems of data sequencing and referential integrity on your own. The tools and techniques for this are unfamiliar to most Salesforce release managers and so have been an afterthought for many teams.

Unfortunately, large managed packages such as Rootstock ERP, nCino, Vlocity, FinancialForce, and Salesforce CPQ require extremely complex configuration data to operate. If you use any of these packages, it's important for your team to establish tools and expertise to facilitate migrating this data, so that you can use your sandboxes to develop this configuration as well. The reason to manage configuration data in your release management process is twofold: first, that configuration data can determine logical pathways that actually affect the behavior of your applications, and second it allows you to develop and test changes to this data before migrating it to production.

To my knowledge, among the packages listed earlier, only Vlocity has built its own configuration data migration tool, Vlocity Build.[11] Prodly Moover has carved out a niche for itself as a tool that specializes in migrating such complex data. Some of the Salesforce release management tools like AutoRABIT, Gearset, and Copado have also begun to develop excellent tools to help in this process. Where these release management tools have an advantage over Prodly is that they can choreograph this data load along with releases. I haven't personally been involved with creating a highly automated configuration data release process from scratch, although it's certainly possible to do so. If you opt to code such a process yourself rather than relying on a commercial tool, here are some suggested steps you'll need to undertake to complete that script:

1. Recognize that data migrations require you to address all the steps mentioned earlier that the Metadata API does for you.
2. Recognize that the skills and experience for doing this will lie with data migration specialists and that there will need to be some logic built into the process as well.
3. Salesforce record IDs and Auto-number fields can vary from org to org, so you can't rely upon those to represent the lookup relationships between objects when you migrate data to other orgs. Therefore, there must be external IDs on all of the objects in the configuration data model so that you can establish uniqueness

[11] https://github.com/vlocityinc/vlocity_build

and define relationships between records in a way that's org independent. Configuration objects in managed packages should have external IDs in place already, but if you've built your own configuration objects, there needs to be at least one required, unique, external ID field on each of them.

4. Assuming the exported configuration data spans multiple objects, the result will be multiple data tables that express their record identity and relationships through external IDs. Those files then need to be loaded in sequence into the target orgs, with parent records being loaded before child records. The Bulk API provides intelligent ways to organize the loading of parent records and child records. But dealing with the Bulk API will require you to do some coding and build some familiarity with its capabilities. The Salesforce CLI introduces the possibility of `data:tree:export` and `data:tree:import` to store and transfer structured data, especially configuration and sample data, using the REST API's Composite resource endpoint. Unfortunately, those commands are limited to 200 records across all trees in one call. This means that to manage large and complex data structures across multiple orgs with minimal overhead, you will need to stitch together many such calls to represent one or more complex data trees. Prepare to do some coding, or retreat to one of the commercial tools mentioned earlier.

5. Retrieving and loading these complex trees from one org to another is clearly a challenge in its own right. If you make careful use of external IDs, marking them required and unique, then you can ensure idempotency any time you load this data in. Idempotency means that if you perform the same actions many times, you will always get the same results. In data migration, this is the same as doing an upsert, matching on external IDs, so that you will create any records that need to be created, but you will never create duplicate records, no matter how many times the data load is run.

6. The only remaining improvement is a performance improvement, so you'll need to assess whether this improvement is even beneficial enough to perform. If you're truly managing large, complex data, it can take a long time to extract, transform, and load that data between orgs. You can make a performance improvement by only retrieving and loading data that has changed since you last checked it. One trick for doing this is to use the REST API's "Get Updated" calls to see if any configuration has been updated since you last checked. This requires tracking and recalling the last time you ran the retrieval or deployment and then generating a coherent set of record updates based on the results from your query. For example, you can run a "Get Updated" call to check each configuration object for record updates since you last ran the command. You can then update the Git repository or database where you're storing this data to track the changes or proceed immediately to deploying those changes. When deploying the changes, you need to query your target org to see if configuration changes have been made in that org. You'll need to have logic to decide whether to preserve the configuration in the target org or in the source org, but if you're driving your configuration through your delivery pipeline, you'll always favor the source org. Based on knowing what changed in the target org, you can know what data to reset to the desired values. Based on knowing what changed in the source org, you'll know what data you need to migrate from source to target. The result of combining these data sets is the data set you need to migrate, which may be significantly smaller than your configuration data set.

This whole process is clearly complicated, which is why I believe this is still a nascent idea among most Salesforce development teams. But the same factors that have driven teams to seek a deployment workflow for metadata will increasingly apply to data in complex orgs. Configuration data changes can be too risky to make directly in production orgs, so they should first be done in development orgs. Those changes sometimes need to be complex, spanning multiple objects, which leads to the need to manage and migrate complex data. This is tedious and error prone for an individual to

migrate; therefore an automated process is important. Therefore building this capability is important to enable our teams to not just configure but to configure safely and efficiently using a migration process to separate development from deployment.

The Security Model

The Salesforce security model is another complex topic to which I'll just offer a brief introduction and some tips to help smooth the development workflow. To make things simple, we can divide this into four topics: infrastructure security, login and identity, admin access, and user access.

Infrastructure Security

Infrastructure security refers to securing the underlying servers, networks, and data stores for an organization. This is a massive area of concern for most IT teams, a massive area of risk and liability for most companies, and one of the important areas that Salesforce handles for you. `http://trust.salesforce.com` can provide you plenty of information on Salesforce's infrastructure security protocols, certifications, and track record, but needless to say, they do an excellent job of handling this for you.

Login and Identity

Login and identity is handled by Salesforce through a variety of mechanisms. Salesforce provides a wide range of Identity and Access Management (IAM) options such as single sign-on (SSO), various types of OAuth, API access, multifactor authentication (MFA), and so on. Whereas infrastructure security is opaque to Salesforce customers, IAM is highly configurable on Salesforce, and customers take joint responsibility to ensure that this is handled correctly. Salesforce does allow you to track and deploy certain aspects of your IAM configuration between environments, and it can be helpful to manage this in your code repository along with other types of org metadata.

Metadata that defines login and identity is shown in Table 4-2. Notice that there's some overlap with the list of metadata in Table 4-1 that you *might not* want to deploy using CI/CD. The reason for this is that this metadata will need to have some variable substitution done if you want to deploy it automatically, and that might not be possible in your first iteration of CI/CD.

Table 4-2. *Types of metadata used to manage login and identity*

Metadata Type	What It Controls and How to Manage It
SecuritySettings	Includes organization-wide security settings such as trusted IP ranges as well as login session settings such as how soon users are logged out after becoming inactive.
ConnectedApp	Defines your OAuth Connected Apps including their callback URLs and consumer keys. The consumer secret is an important part of this definition, but is not stored in metadata for security reasons.
AuthProvider	Defines social sign-on providers like OpenIdConnect and Facebook that can be used to authenticate users. Social sign-on is typically used to authenticate Community users, as opposed to regular Salesforce users, although OpenIdConnect is becoming increasingly common in the enterprise. For example, Microsoft Azure Active Directory is a cloud-based identity provider which uses OpenIdConnect for SSO.
SamlSsoConfig	Defines your SAML SSO configuration, including the login URLs and validation certificate. SAML 2.0 has traditionally been the enterprise standard for SSO, but OpenIdConnect is increasingly common for enterprise use.

Some aspects of login and identity security are also handled by Profiles, and Login Flows can also affect this. Profiles can be used to specify login IP ranges and times on a per-user basis. Login Flows provide sophisticated authentication options such as calling out to third-party authentication services and assigning temporary session-based permissions.

For teams that are just getting started tracking their Salesforce code and configuration in version control, it may seem like overkill to track this kind of metadata in the repository. In fact in some cases, this metadata needs to vary slightly for different environments. For example, Connected Apps in sandboxes might have different endpoints that point to sandbox versions of other systems. But it is precisely because this configuration is so important that it makes sense to track and deploy it from version control. Many changes to login and identity should be tested in a sandbox first, and in any case, it's extremely helpful to see a history of changes. An errant admin could readily break integrations, but having version control provides you the ability to roll back and monitor changes. Automating the deployment of some of these metadata types requires

that you have the ability to dynamically substitute values as part of your deployment process. This is a native capability of tools like Copado, but we provide an overview of how you can build this yourself in "Managing Org Differences" in Chapter 9: Deploying.

Admin Access

Admin access refers to ensuring that only the appropriate people have permission to make certain changes in your org. Remember that everything in Salesforce is defined by configuration and that even custom code on Salesforce is a form of configuration. You can think of admin access as configuration security, or "what you can/can't do in the Salesforce Setup UI."

Salesforce provides a built-in System Administrator profile which possesses "God Mode" privileges in your org. Needless to say, you should be very selective in who is given this profile. Most users would be assigned the out-of-the-box "Standard User" profile, which typically provides the ability to use Salesforce and see data, but not to change configuration.

Admin access and user access are the two areas where admins have the most ability to tune Salesforce to meet their organizational needs. Most organizations will eventually find the need to make a copy of the System Administrator profile and the Standard User profile and begin to customize these according to the needs of their organization.

The traditional method of providing access to control configuration has been through the use of Profiles. But as mentioned earlier, Profiles are a nightmare to manage in version control and typically don't provide the right level of granularity that teams need. Permission Sets should be your preferred method to manage permissions. You should keep profile permissions extremely thin and instead use permission sets to determine both "admin"- and "user"-level access. If possible, limit your use of profiles to defining things like login IP ranges which can't be defined in Permission Sets. More details on this are provided in Chapter 11: Keeping the Lights On.

User Access

Whereas admin access determines who can do what in the Salesforce Setup UI, user access refers to what end users can access or modify inside the Salesforce application itself. Data security defines which users have create, read, update, or delete (CRUD) permissions on Salesforce data objects. User access also defines who can use particular Apex classes, applications, Visualforce pages, and standard Salesforce applications like

Service Cloud and Knowledge. Permissions in Salesforce are always **additive**, meaning that the default is for users to have no permissions, but they gain permissions as a result of their profile, user settings, permission sets, and permission set licenses.

As mentioned, all of the admin and user access privileges are defined using profiles or (preferably) permission sets. Permission sets can and should be tracked in version control and deployed as part of your release management process. This provides history tracking on changes to these and ensures that permissions can be tested thoroughly before being deployed to production.

One of the most common failure modes for custom Salesforce applications is for developers to not deploy appropriate permissions along with other metadata. Developers can and should have System Administrator privileges in their own development environment. Such unobstructed access allows them to build capabilities without restriction. But it also makes it easy for developers to forget that end users will need explicit permissions to use what the developer has built.

Developers and testers should use the "Login As" capability to log in to their development and testing environments as one of the target users to ensure those users will have access. If a developer creates a custom application, a new object, with new fields, a tab for that object, a Visualforce page, and an Apex controller for that page, users will need permissions for each of those components. Thus as the complexity of an org increases, so too does the complexity of user permissions.

If you use profiles to manage permissions and you want to give all profiles access to this custom application, then you will need to assign those permissions to every profile. Permission sets thus help you follow the DRY maxim, "don't repeat yourself." You can define permissions for this new custom application in a single permission set and then assign that permission set to all users in the org.

Profiles and Permission Sets are both metadata and can be deployed between environments. But which users are assigned those profiles and permission sets cannot be deployed in the same way. This means that when deploying a new permission set to production, you will either need to manually assign this to all users or preferably write a bit of anonymous Apex to query all User IDs and create PermissionSetAssignment records for each user to assign the new permission set to them.

Salesforce is working on the concept of "Permission Set Groups," which is currently in Pilot. This provides a much more elegant alternative. You attach multiple permission sets to a permission set group and then assign a large group of users to that single Permission Set Group. Permission set groups provide a simple way to bulk assign

permissions consistently to a group of users. They can be updated by adding or removing permission sets to a group, and those permission sets can be used across multiple permission set groups.

Permission Set Groups are not accessible via the Metadata API in the pilot, and so you would still need to add any new permission sets to the group manually. If this is eventually made accessible through the Metadata API, then developers will have the ability to create new permission sets in a development environment, add them to the appropriate permission set group, and deploy those permissions all the way to users in production.

A new type of Permission Set called a "Muting Permission Set" is also in pilot. Muting Permission Sets can be added to a Permission Set Group and provide the ability to "Mute" permissions. To my knowledge, this is the first example of Salesforce enabling a **negative** permission on the platform. The explicit purpose of this is to inhibit permissions that would otherwise be given to members of the group by other Permission Sets in that same group. It might be possible to use this for security purposes to ensure that members of the group are never given powerful admin privileges such as "Modify All Data," but this use case is not mentioned in the documentation.

Code-Based Development on Salesforce

Salesforce allows for both server-side programming and client-side programming on Salesforce. Apex triggers, Apex classes, and Visualforce are all server-side programming options. Although Visualforce pages include dynamic JavaScript, they are compiled and sent from the server like PHP. Client-side programming includes Lightning Web Components and Lightning Aura Components, but might also include the use of complex JavaScript inside of Visualforce.

This section is very brief, since there are so many other references on how to write high-quality code for Salesforce. You should also refer to the discussions about static analysis and unit testing in Chapter 8: Quality and Testing and to the discussion of "Monitoring and Observability" in Chapter 11: Keeping the Lights On.

Server-Side Programming

Server-side programming in Salesforce allows direct access to the underlying Salesforce data model, and a large part of its purpose is to allow for queries, processing, and transformation on this data that cannot be done through clicks alone. Visualforce provides the ability to build custom user interfaces, although Lightning Web Components are now the preferred method to do this.

Apex

Apex is a strongly typed language, similar to Java, that provides native access to some underlying Salesforce capabilities. The most obvious native capability in Apex is the ability to work with Salesforce data as first-class Objects. So, for example, you can perform a query on Accounts and then perform actions on that data without having to explicitly define an "Account" object. Although Apex compiles down to Java behind the scenes, its most notable limitation is that you can't use third-party Java libraries or some advanced features of that language.

Salesforce runs Apex on the core platform, and Apex can't be compiled or run outside of a Salesforce org. The limitations on the language are largely to ensure that it can run safely in a multitenant environment without causing excess load or accessing things that would compromise the security of the platform.

There are actually two kinds of Apex metadata: triggers and classes. Triggers are a concept borrowed from other relational databases and allow custom code to be run as part of a database transaction (insert, update, delete, or undelete). Apex classes are more flexible than triggers, since they allow the use of methods, interfaces, global variables, and so on. Triggers were the first form of custom code allowed on the platform and have a very strictly defined format that does not allow the use of methods or some other capabilities of Apex classes.

Salesforce executes Apex in the context of a transaction. To ensure that no single transaction goes out of control and soaks up excessive system resources, Salesforce strictly enforces governor limits on heap size, CPU time, number of SOQL and DML statements, and so on. These governor limits are uncomfortable for developers accustomed to running code on their own servers, but are a design constraint that encourages code to be performant and balances freedom with the needs of other users on the platform.

Each Apex class or trigger runs using a specified version of the Salesforce API. This wise design decision allows Salesforce to make breaking changes in a new version of

Apex without impacting code that was written previously. When downloaded, this API version is shown in a "sidecar file" that has the same name as the class or trigger with an added `-meta.xml` suffix. As a best practice, classes and triggers should periodically be reviewed and upgraded to the latest API version to ensure optimal runtime performance. In most cases, updating the API version is a trivial change, but it's possible to face compilation errors, and this is a good reason to write good quality Apex unit tests to ensure behavior remains unchanged even if you update the API version.

It is essential to avoid hardcoding IDs in Apex code or anywhere else. Record IDs are typically different in each environment. Instead, ensure your logic can dynamically identify the proper data to operate against and not fail.

Visualforce

Visualforce allows you to create completely custom user interfaces, using an HTML-like syntax to display information and invite user input and actions. Visualforce also excels in giving access to Salesforce's internal data model through the use of Standard Controllers or Controller Extensions. A Visualforce page that uses a Standard Controller for Account allows you to create, access, and update Accounts without any other backend code. This page can override the standard record detail or record edit page in Salesforce, thus providing an alternative user interface for working with any Salesforce object.

There are many other capabilities of Visualforce, but it's fallen out of favor since it's relatively slow compared to Lightning-based pages. Visualforce pages have to be rendered on Salesforce, and their state is transferred back and forth from the browser each time you perform an action or move to a new page. There are also strict limitations to the size of this Viewstate that make them unsuitable for editing large amounts of data.

Scripting and Anonymous Apex

There are other ways to execute Apex such as creating custom REST and SOAP services. But one form of Apex worth mentioning here is "anonymous Apex," which is not stored on the server but can nevertheless be run on demand.

Among the many uses of anonymous Apex is the ability to automate predeployment and postdeployment steps in the process of building and delivering an application. Anonymous Apex has the ability to query, transform, and update data, and this includes system-level data such as permission set assignments.

As mentioned earlier, governor limits make anonymous Apex unsuitable to perform massive or long-running data transformations such as migrating data between fields on more than 10,000 records at a time. For this purpose, you can use batch Apex. But there are many practical one-time activities that you can perform using anonymous Apex, such as creating or removing scheduled jobs, triggering batch Apex to run, modifying User Role assignments, and so forth.

Although anonymous Apex is not persisted inside of Salesforce, you can and should save such scripts in your code repositories. This allows for reuse and also provides a clear audit trail should you or someone else need to review the changes that were made.

Client-Side Programming

Client-side programming involves systems where the bulk of processing and state management happens inside the user's client, either a web browser or the Salesforce mobile app. Lightning Web Components are now the recommended way of creating custom user interfaces, and they provide many benefits over Visualforce. Salesforce's original Lightning Component technology was based on an open source framework known as Aura. Although it's a bit confusing and frustrating for developers to have to learn new technologies, it's par for the course, and there are good reasons why Lightning Web Components have come into being. There are also a variety of other ways to connect to Salesforce from a web client, such as JavaScript Remoting.

Lightning Web Components

Most people who were using the Internet in the late 1990s and early 2000s were aware of the "Browser Wars," when Internet Explorer competed with Firefox and eventually Chrome for market dominance. But only web developers are familiar with the "Framework Wars" that pitted Angular against React and a hundred other frameworks to provide a more robust way to build applications on the Web.

Web Components are a standards-based alternative to custom JavaScript frameworks that are natively supported by modern web browsers. Unlike custom frameworks like Angular and React, Web Components provide native execution that allows them to run quickly and a consistent syntax that allows developers to reuse their skills more easily. Salesforce was a participant in developing the open standard for Web Components, and Lightning Web Components are an implementation of the standard that is optimized for working with Salesforce.

Lightning Web Components were announced in late 2018, but Salesforce had been quietly rewriting their entire Lightning user interface in LWC for a year or more. This enabled much faster performance and gave Salesforce the confidence that it was possible for customers to mix Lightning Components and Lightning Web Components together on the same page. Salesforce themselves had been refactoring the application in that way for over a year!

Lightning Aura Components

The original Lightning Component framework was based on an open source project called Aura, inspired by AngularJS. Like other code built on custom frameworks, Lightning Aura Components require the browser to do more work, since they have to be compiled into native JavaScript, adding significant amounts of execution overhead.

The vision for Lightning Aura Components is powerful and inspiring, since it allows organizations to build custom UI components using the same technology used to create Salesforce's own UI. Salesforce Classic is a fundamentally different technology than Visualforce. And Visualforce pages actually run in a separate domain to ensure transaction security and prevent custom Visualforce code from scraping data that it should not have access to.

By creating Lightning Aura Components, Salesforce opened the door to a long-term vision in which developers could seamlessly mix custom components with built-in components, even down to overriding a single field. That vision has not yet been fully realized, and Salesforce is still making incremental improvements to balance flexibility with security. But the Salesforce UI is far more responsive and performant today than before Lightning was rolled out.

JavaScript Remoting, S-Controls, and API Access

There are other client-side coding options on Salesforce that predate Lightning Components. Three official options are JavaScript Remoting, Visualforce Remote Objects, and S-controls. JavaScript Remoting is a technique that allows Visualforce pages to host complex JavaScript applications that can store state on the client side while still sending and receiving data from Salesforce using an Apex controller. Visualforce Remote Objects allow you to create a JavaScript representation of Salesforce objects so you can create, retrieve, and update Salesforce data using client-side JavaScript. S-controls were deprecated many years ago, but were Salesforce's first foray into allowing custom coding.

They allow you to create custom HTML and JavaScript in an iFrame, which can access Salesforce data using the standard Salesforce API.

Salesforce's API is also what enables a variety of other third-party custom code solutions such as Restforce for Ruby and Simple Salesforce for Python. These prebuilt libraries provide convenient wrappers around the REST API that allow teams familiar with those languages to work with Salesforce data and trigger Salesforce operations. The most significant such library is JSForce for JavaScript. JSForce is now the engine used by the Salesforce CLI to communicate with Salesforce and is also at the heart of many other JavaScript, TypeScript, and Node.js libraries for Salesforce.

Summary

This has been a brief introduction to the process of developing on Salesforce. The Salesforce DX team is responsible for a huge portion of the Salesforce platform, including developer tooling. This developer tooling has been evolving rapidly since the introduction of Salesforce DX, although there are still some key improvements needed. While most Salesforce developers focus on the actual technologies used to build on the platform with clicks and code, we introduced key concepts about the Metadata API that are important for facilitating the entire DevOps process.

We gave a brief summary of the click-based and code-based options for building on the platform, but we intentionally focused on those aspects most relevant to automating deployments to other orgs.

Much has been written about developing on Salesforce, but far less has been said about how to make the release management process easier. Perhaps that's because the people with the knowledge and skill to manage that process are too busy doing releases to take the time to share that knowledge. Or perhaps they're so burned out by following manual processes that they don't want to think or talk about the subject any more. I've somehow been fortunate enough to gain release management experience and live to tell about it, especially about how the process can be made easier.

Hidden in between the topics of building and releasing is the topic of architecture. It's fair to say that you must control your architecture, or it will control you. Salesforce makes it easy to build, which implies that it also makes it easy to build badly. If you're just beginning down the path of configuring Salesforce, now is the time to study carefully the techniques in the next chapter so you can build your org in a scalable and modular way. If that time has long since passed, then hopefully the techniques in the next chapter can provide you ideas on how to dig yourself out of technical debt.

CHAPTER 5

Application Architecture

In the business world, it's common to distinguish between strategy and tactics, two terms borrowed from the military. Strategy refers to high-level decisions that fundamentally determine the landscape in which you operate. In military planning, strategy dictates where to move armies, how to attract and train armies, what kinds of weapons to invest in, and so on. In business planning, strategy dictates what markets to pursue, whether to invest in onshore talent or outsource, what new products to develop, and so on. By contrast, tactics dictate how to succeed on a small scale. In the military, tactics refers to the skills and tools needed for an individual to survive and win in activities such as hand-to-hand combat. In the business world, tactics address topics such as how to organize teams, motivate employees, communicate to customers, and so on.

This distinction extends to the world of Salesforce DevOps. Part 1 of this book speaks to the overall strategy you can apply to drive innovation and continuous improvement for your organization using Salesforce. Parts 3 and 4 of this book speak to the tactics of setting up a delivery pipeline as the mechanism to deliver innovation safely and get feedback from production.

Although Part 2 of the book (you are here ⬇) covers developing on Salesforce, I've said very little about how to actually develop on Salesforce. How to develop on Salesforce is a **tactical** discussion. It's a complex topic, but one that is very well covered in Salesforce's documentation and endless other books, blogs, and talks. What is not as commonly understood is how to architect Salesforce applications in such a way that they allow for continual innovation and lend themselves to being packaged and deployed independently. On the level of coding, this is a **strategic** topic.

Martin Fowler described architecture as "those aspects of a system that are difficult to change." There's not a clear-cut distinction between a developer and an architect, but the expectation is that an architect has sufficient experience and understanding to make decisions at the outset of a project that the team will not come to regret. Just as the role of a military general is to get armies to the correct battlefield so they don't waste their effort and skill fighting the wrong battles, the role of a Salesforce architect is to be a strategist

A. Davis, *Mastering Salesforce DevOps*, https://doi.org/10.1007/978-1-4842-5473-8_5

and hold a wise vision of the system being designed so that development teams can build and deploy their applications in a scalable way.

The *State of DevOps Reports* identify application architecture (especially loosely coupled architecture) as one of the key enablers of continuous delivery and thus one of the key drivers for delivering value to the organization.

We'll begin by looking at modular architecture and how dependencies arise and then look at various techniques for modularizing your code.

The Importance of Modular Architecture

> *The architecture of your software and the services it depends on can be a significant barrier to increasing both the tempo and stability of the release process and the systems delivered. ... We found that high performance is possible with all kinds of systems, provided that systems – and the teams that build and maintain them – are loosely coupled. ... The biggest contributor to continuous delivery in the 2017 [State of DevOps] analysis – larger even than test and deployment automation – is whether ... the architecture of the system is designed to enable teams to test, deploy, and change their systems without dependencies on other teams. In other words, architecture and teams are loosely coupled.*[1]
>
> *—Accelerate: Building and Scaling High Performing Technology Organizations*

A vivid indication of the impact of modular architecture on team performance can be inferred from Figure 5-1. This graph, taken from the *2015 State of DevOps Report*, shows a very surprising effect. In general, as the size of a development team increases, the traditional expectation is that the number of deploys per day would decrease due to the increasing complexity of the application. That trend is seen in the "Low Performers" trend line, which shows deployments per developer decreasing as the team size approaches 100 people. What is surprising is that "High Performers" (the high-performing software organizations mentioned previously) actually record a dramatic increase in the number of deployments per day per developer as the team size increases. Such performance increases are only possible with a loosely coupled architecture, one that is free from complex interdependencies between teams and packages.

[1]Nicole Forsgren, Jez Humble, and Gene Kim, *Accelerate: The Science of Lean Software and Devops Building and Scaling High Performing Technology Organizations* (IT Revolution Press, 2018), 76.

Figure 5-1 doesn't depict data for low-performing teams with more than 100 people, and it doesn't depict data for medium-performing teams of 1,000 people. This is presumably just a gap in their sample data, but if you extrapolate those curves, it hints at a possible conclusion: it may be impossible to scale your team beyond a certain size unless you have enabled that team to deliver software effectively. DevOps is sometimes felt to be easier to do at smaller scales. It's certainly easier to implement new processes while you're small. But this graph may be indicating that it's only through implementing DevOps principles that you'll *be able* to scale your business without sinking into crippling inefficiencies.

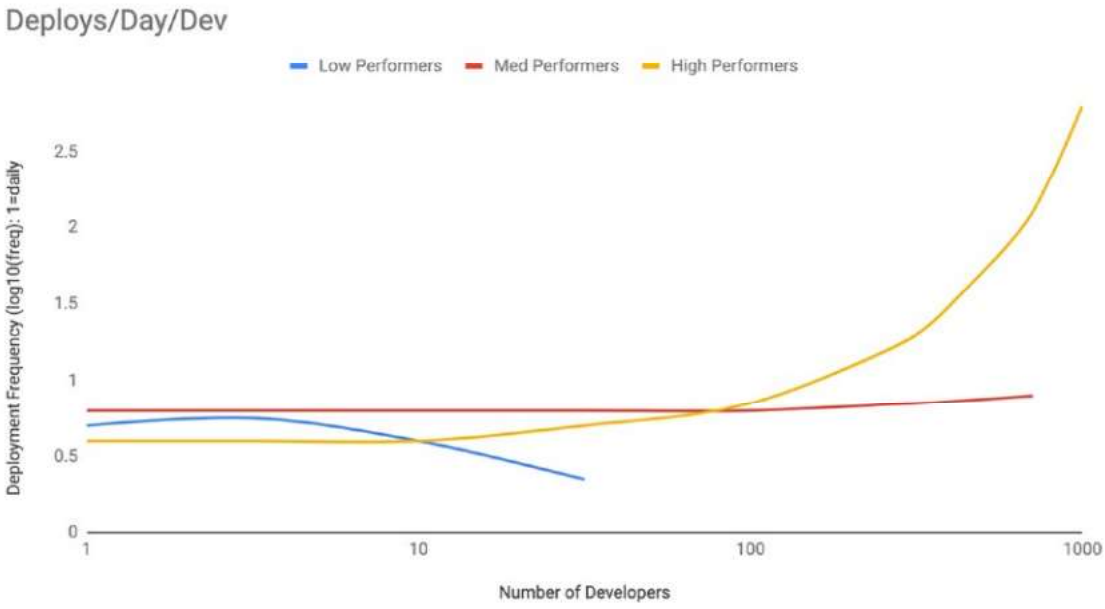

Figure 5-1. *Number of deployments per day per developer. 2015 State of DevOps Report*

Understanding Dependencies

Salesforce metadata, like any system, is interconnected with other components. Each time a piece of metadata references another piece of metadata, it establishes a dependency on that component. The net result of this is a tangled web of dependencies like that shown in Figure 5-6. Such a dependency network is known as "tight coupling," where different parts of the system have definite dependencies on other parts of the system. The opposite of this is "loose coupling" in which multiple systems can interact with one another but don't depend on one another.

Tightly coupled architecture is a characteristic of many systems, not only Salesforce, that have evolved over time without taking care to avoid this risk. One famous example and counterexample was given by Steve Yegge[2] and widely shared by Jez Humble and others in the DevOps community. Yegge's "Platform Rant" contrasted Google's architecture with that of Amazon, both his former employers. While clearly complimentary to Google, Steve Yegge pointed out that Amazon did an extraordinary job of enforcing strict separation between their systems with communication only through published APIs. By contrast, Google maintains a single code repository containing 2 billion lines of code and 86 TB of data,[3] within which there is little or no isolation between sections of code.

Google has made this work through an extraordinary team of engineers and highly customized version control and developer tooling systems. But deploying and testing such complex interdependency is a fundamentally hard problem to solve. Amazon's architecture, with each team enforcing segregation from all other teams and communicating only through APIs, has presumably made it much easier for each of their teams to evolve their systems independently of one another, but it's also led to an extremely important byproduct: AWS.

Amazon Web Services is the leading cloud infrastructure provider by a large margin. They support the computing infrastructure for more than a million businesses[4] and control one third of the cloud infrastructure market.[5] Their customers include Salesforce and the US Federal Government. AWS is built on the same loosely coupled architecture that was built to support their eCommerce business. Because each AWS service is independent, they each have published APIs, can be billed and provisioned separately, and interconnected to each other or to external services with almost equal ease.

Google and Amazon are dramatic (and dramatically different) examples. There are benefits and tradeoffs to both a tightly coupled and a loosely coupled approach. But it is important to recognize that although superficially it may look like Google has a tightly coupled architecture, they do not. Their underlying tooling allows their 25,000 active developers to collaborate on a common trunk, but commits can be automatically rolled back, dependencies can be automatically analyzed, regression testing is performed

[2] https://gist.github.com/chitchcock/1281611

[3] https://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext

[4] https://expandedramblings.com/index.php/amazon-web-services-statistics-facts/

[5] www.canalys.com/newsroom/cloud-market-share-q4-2018-and-full-year-2018

automatically, and so forth. So in effect, Google achieves loose coupling on the fly through the miraculous use of tooling automation.

Barring some miraculous, not-yet-invented tooling, your fastest path to delivering innovation quickly and safely to your users will be to try to constrain the interdependencies in your system so that portions of the codebase maintained by different teams are not tightly dependent on one another. This is the meaning of building a modular, loosely coupled architecture.

Many of the following techniques are focused on Apex classes and triggers, since these are the most complex type of metadata and the ones which provide the most creative options for loose coupling. Where possible, I've offered suggestions for other metadata types as well.

Salesforce DX Projects

A discussion of Salesforce application architecture begins with a look at Salesforce DX projects, since it's these projects that open the door to building with a modular architecture.

How Salesforce DX Projects Enable Modular Architecture

The Salesforce DX project structure was introduced earlier, but there are a few aspects of this project structure that are worth looking at in more detail, since they enable special capabilities.

Salesforce faced a major challenge when envisioning Salesforce DX as a new way of building and managing applications. On one hand, they needed to ensure backward compatibility with teams using legacy tools. They also needed to accommodate customers whose development teams used a mix of old and new tools. They were also aware of the cost and complexity of building and testing entirely new APIs for managing application metadata.

One of the first big challenges was how to enable teams to store metadata in folders. Folders are not exactly a novel concept in computer operating systems, but the concept was never built into the Metadata API, since it was not built to handle source-driven development. What the Salesforce DX team came up with was a compromise that

continues to use the Metadata API to communicate with Salesforce, but allows teams to divide metadata into folders on their local development environment and route metadata changes back and forth between those local folders and their development org.

Those folders allow teams to begin to group related metadata together. This provides information on which metadata components are more closely related, but can also be used to segregate areas of responsibility between different teams. More importantly, those folders form the basis for being able to build multiple packages in a single code repository.

Scientists hypothesize that the earliest single-celled organisms began as organic molecules isolated inside a sphere of phospholipids that separated their contents from the water surrounding it. Gradually this lipid sphere became the plasma membrane that surrounds complex cells capable of reproduction. Salesforce DX folders are like this lipid sphere that provides a gentle division between certain groups of metadata. Packages are the full evolution of this division that allow groups of metadata to become autonomous units that can be deployed and upgraded on their own.

The key file that defines a Salesforce DX project is the `sfdx-project.json` file contained in the project root. When an SFDX project is first created, this file specifies a single default package folder called `force-app/` and creates metadata in `force-app/main/default`. Importantly, you can immediately begin segregating metadata into different subfolders inside of `force-app/` even without changing the `sfdx-project.json` file.

The `sfdx-project.json` file becomes more important when you begin to define actual second-generation packages inside your code repository. In this case, each package must be associated with a specific folder, and there should be no overlap between these package folders. For example, you might define a package called `main` that is associated with the folder `force-app/main/default` and a second folder called `sales` associated with the folder `force-app/sales`. One package must remain the "default" package, since any newly created metadata will be downloaded and stored in this default package until it's moved elsewhere.

The Package Directories section in `sfdx-project.json` allows you to specify dependencies, either on other packages in the same project or on packages defined elsewhere. They can also specify environment requirements by linking to a scratch org definition file.

Scratch org definition files are another important feature of the Salesforce DX project format. They specify the characteristics of a scratch org and are used in the package publishing process to explicitly state the edition, features, and settings that must be in place for the package to work.

To evolve your Salesforce org configuration from an unstructured collection of metadata (the "happy soup") to a well-ordered set of interdependent packages is one of the biggest architectural challenges on Salesforce today. We can perhaps take inspiration from the evolution of life on earth, where it took 750 million years for single-celled organisms to evolve from a primordial soup. With packaging we do have the benefit of intelligent design, including the techniques shared in the remainder of this chapter. Even if it takes your team a few years to accomplish, you will be well ahead of Mother Nature.

Creating a Salesforce DX Project

If you're starting a new project, you can start using Salesforce DX from the very beginning. Developing using scratch orgs allows you to develop in your own private environment while easily integrating changes from the rest of the team. Dividing your codebase into unlocked packages allows separate teams to develop and release independently, with confidence that every aspect of the application is included.

For teams moving an existing org over to Salesforce DX, there are two main challenges to be overcome. First of all, it's difficult to untangle all of an org's metadata into logical packages. Second, complex orgs defy attempts to recreate all the underlying dependencies in scratch orgs. Much of the rest of the chapter talks about techniques that can help you to decouple dependencies to allow you to build discrete packages. But this undertaking requires time and care, so what you'll find here are suggestions and guidelines rather than simple solutions.

You'll first want to decide on which Dev Hub to use, and enable access for all collaborators. See "The Dev Hub" in Chapter 6: Environment Management. In general, it's important for the whole team to use the same Dev Hub so that they can all access and update the packages created during the development process, as well as capabilities such as org shapes, namespaces, and scratch org snapshots, all of which are stored on the Dev Hub.

The Salesforce CLI provides a command `sfdx force:project:create` that creates or "scaffolds" the basic files needed to use Salesforce DX. You can use that folder structure as a starting-off point. If you have been tracking existing metadata using version control, you may want to see the section "Preserving Git History When Converting to Salesforce DX" in Chapter 7: The Delivery Pipeline for tips on converting without losing history.

As discussed in Chapter 7: The Delivery Pipeline, there are different branching strategies and different CI/CD jobs for managing org-level metadata compared to

managing Salesforce DX package metadata. For this reason, your life will be far simpler if you have at least two separate code repositories: one for org-level metadata and one for packaged metadata. You can then have separate code repositories for each package, which may simplify the package publishing process. With a bit of automation, you can also combine multiple packages into a single repository as described in the same chapter.

I recommend you create a single parent folder to hold these multiple repositories on your local machine and a team/group/project in your version control system to give everyone on the development team access to all these repositories. It can be tempting to want to strictly segregate the codebase between different teams, but this generally just causes confusion and inefficiency. Salesforce metadata can interact in complex ways. It's important for anyone on any of your teams to be able to see what code and configuration has been defined, so they can reuse that, interface with it successfully, and hopefully help with refactoring across package boundaries as your team's needs evolve. If you don't trust someone enough to give them access to your organization's codebase, you probably shouldn't have hired them. If you want your team to be efficient and collaborative, you'll need to promote trust as an internal core value. Version control enables tracking who made which changes and rolling back mistaken changes.

Part 3: Innovation Delivery details how to create a continuous integration system that will be used by your entire team for testing and deployments. When you set up your local development environment, you'll connect to the Dev Hub and development environments using your personal credentials. When setting up the CI systems, you should use the credentials for an integration user. Providing your personal credentials to set up a CI system may constitute a security risk in some cases or cause a service interruption if your personal user account is deactivated. Using an Integration user to run automated jobs on each Salesforce org in your delivery pipeline avoids these issues.

Modular Development Techniques

So what are some of the modular development techniques that can be employed on the Salesforce platform? We first look at some oldies but goodies, before moving to some of the newer and more powerful techniques.

Naming Conventions

Naming conventions are a "poor man's packaging." Nevertheless, they have been one of the few available methods to segment metadata in Salesforce and thus deserve recognition. Martin Fowler famously remarked that "Any fool can write code that a computer can understand. Good programmers write code that humans can understand." Naming things is a big part of that, and naming Classes and other metadata begins to bring sense and structure to an otherwise chaotic system.

Naming conventions in Salesforce generally take the form of prefixes and suffixes. Prefixes include the use of "fake namespaces" that often represent a department or group of capabilities. Salesforce packages can take advantage of real namespaces like `SBQQ__` that are separated from the metadata name by two underscores. But any team can create fake namespaces like `HR_` separated from the metadata name by a single underscore. As you begin to move your organization's metadata into Salesforce DX package folders, you may be able to use such fake namespaces as clues left by previous developers about which metadata belongs together.

The use of suffixes in metadata names is typically to indicate the specific function of a Class rather than which application it belongs to. By convention Apex tests should have the same name as the classes they test, but with the suffix `Test`. Visualforce controllers will ideally have the suffix `Controller`; Schedulable Apex will ideally have a suffix `Schedule`; and so forth.

None of these prefix or suffix conventions are enforced, but it is a good convention for your team to adopt and adhere to wherever possible.

Object-Oriented Programming

Object-oriented programming is in fact an example of modular design. Fortunately, this capability is baked into Apex and is already in widespread use, but it's worth recalling its benefits. The purpose of object-oriented design is to centralize data and capabilities inside objects, to promote interaction between objects as opposed to disconnected variables and functions, and to allow for sophisticated techniques such as inheritance.

Apex classes are just that—they are object-oriented classes that are usually instantiated into objects. Their functions are actually methods attached to the class, the variables defined in those classes are local variables by default, and so on. Teams who

are already making wise use of Apex's object-oriented architecture will find it easier to adopt some of the techniques described later, such as dependency injection, packaging, and separation of concerns.

An important aspect of object-oriented programming that's not commonly used by most Salesforce programmers is the use of inheritance and interfaces to create classes that share common characteristics. Inheritance is a requirement for using Dependency Injection, since it allows you to write code based on the generic behavior of classes, even if you don't know which specific class will be used when your code is run.

Dependency Injection

Dependency Injection (DI) is a sophisticated programming technique that began to gain significant attention when Salesforce DX introduced unlocked packages. DI truly allows for modular architecture by allowing you to define dependencies at runtime, as opposed to when the code is compiled.

The example used here is borrowed from Philippe Ozil's excellent blog post "Breaking Runtime Dependencies with Dependency Injection,"[6] and I refer you to that post for a more clear and detailed explanation. Other seminal contributions on this topic come from Andrew Fawcett,[7] Douglas Ayers,[8] and Jesse Altman,[9] as well as from John Daniel, the technical reviewer for this book.

Normally, when you make a reference from an Apex class called `MyClass` to another class, you do so by specifying the other class' name and the method you want to access such as `OtherClass.doSomething()`. When you save your class, the Apex compiler checks to ensure that there is in fact a class called `OtherClass` that contains a method called `doSomething`. This establishes a hard dependency from `MyClass` to `OtherClass`.

Normally this is not a problem, but it leads to several limitations. First, there is no flexibility built into that approach: `MyClass` simply depends on `OtherClass` to execute some processes. This means that the behavior cannot vary per environment, be configured at runtime, behave differently as part of an Apex test, and so on. Second, it

[6] https://developer.salesforce.com/blogs/2019/07/breaking-runtime-dependencies-with-dependency-injection.html

[7] https://andyinthecloud.com/2018/07/15/managing-dependency-injection-within-salesforce/

[8] https://douglascayers.com/2018/08/29/adopting-dependency-injection/

[9] http://jessealtman.com/2014/03/dependency-injection-in-apex/

means that `OtherClass` must be present in the environment when `MyClass` is saved. This can prevent the two classes from being separated into different packages that can be deployed independently.

Ozil's article shows how to use the concept of Inversion of Control to access other classes indirectly instead of hardcoding references directly. Listing 5-1 shows a code snippet in which `OrderingService` directly depends on `FedExService`. Inversion of Control adds a layer of abstraction to return the `ShippingService` indirectly as shown in Listing 5-2. This gives additional flexibility at the expense of some added complexity.

Ozil then shows how this code can be refactored further so that the dependencies can be determined at runtime. Dependency injection uses the Apex `Type` class to dynamically create an instance of the appropriate ShippingService as shown in Listing 5-3. That example shows how a ShippingService class can be specified at runtime using Custom Metadata.

Using Custom Metadata to specify dependencies dynamically is a very robust solution and one that is used by the Force-DI package[10] cocreated by Andrew Fawcett, Douglas Ayers, and John Daniel. Custom Metadata can be created and deployed from a development environment to production, but can also be overridden in a particular environment if appropriate. Custom Metadata records can also be distributed across multiple packages as long as the Custom Metadata type is defined in a common package dependency.

The Force-DI package supports dependency injection in Apex, Visualforce, Lightning, and Flows, and can be used as a foundation for your own code. The package defines a Custom Metadata type to store references to the metadata you want to call at runtime. Where you want to use dependency injection, you pass an instance of the `di_Injector` class as a parameter and use that to access the metadata that has been configured for that org.

One of the most broadly applicable use cases for this is when creating Triggers that rely on code from multiple unlocked packages. This use case is discussed in the section "Modular Triggers with Dependency Injection".

[10] https://github.com/afawcett/force-di

Listing 5-1. OrderingService has an explicit dependency on FedExService (from Philippe Ozil's blog post)

```
public class OrderingService {

  private FedExService shippingService = new FedExService();

  public void ship(Order order) {
    // Do something...

    // Use the shipping service to generate a tracking number
    String trackingNumber = shippingService.generateTrackingNumber();

    // Do some other things...
  }
}
```

Listing 5-2. OrderingService refactored to use inversion of control to determine the ShippingService indirectly (from Philippe Ozil's blog post)

```
public class DHLImpl implements ShippingService {
  public String generateTrackingNumber() {
    return 'DHL-XXXX';
  }
}

public class FedExImpl implements ShippingService {
  public String generateTrackingNumber() {
    return 'FEX-XXXX';
  }
}

public class ShippingStrategy {
  public static ShippingService getShippingService(Order order) {
    // Use FedEx in the US or DHL otherwise
    if (order.ShippingCountry == 'United States') {
      return new FedExImpl();
    }
```

```
    else {
      return new DHLImpl();
    }
  }
}

public class OrderingService {
  public void ship(Order order) {
    // Do something...

    // Get the appropriate shipping service
    // We only see the interface here, not the implementation class
    ShippingService shipping = ShippingStrategy.
    getShippingService(order);

    // Use the shipping service to generate a tracking number
    String trackingNumber = shipping.generateTrackingNumber();

    // Do some other things...
  }
}
```

Listing 5-3. OrderingService refactored to use dependency injection to determine the behavior at runtime (from Philippe Ozil's blog post)

```
public class Injector {
  public static Object instantiate(String className) {
    // Load the Type corresponding to the class name
    Type t = Type.forName(className);
    // Create a new instance of the class
    // and return it as an Object
    return t.newInstance();
  }
}

// Get the service implementation from a custom metadata type
// ServiceImplementation.load() runs a SOQL query to retrieve the
   metadata
Service_Implementation__mdt services = ServiceImplementation.load();
```

```
// Inject the shipping service implementation
// (services.shipping is either FedExImpl, DHLImpl or any other
implementation)
ShippingService shipping = (ShippingService)Injector.
instantiate(services.shipping);

// Use the shipping service to generate a tracking number
String trackingNumber = shipping.generateTrackingNumber();
```

Event-Driven Architecture

Event-driven architecture is an ultimate example of loosely coupled architecture. In this model, components communicate by passing events as opposed to being directly connected to one another. One system will **publish** an event, and other systems can **subscribe** to categories of events that they're interested in. For this reason, it's sometimes also called a "pub-sub architecture." Typically, there's also a common **event bus** where those events are stored for a period of time. The event bus provides a shared platform for event publishers and subscribers and can provide capabilities like unique message IDs, sequencing, and event playback.

As an analogy, I can subscribe to an email list to receive information about webinars I might be interested to attend. My relationship with those webinars is "loosely coupled": if I don't attend, the webinar will still go on; and if they don't have the webinar, my life will still go on. I can subscribe or unsubscribe based on my interests and take action as appropriate.

The first event-driven architecture to become popular in Salesforce was Lightning Events, introduced in 2015, which provide a way to pass messages between Lightning Components in the same browser window. This architecture means that you can have one Lightning Component that responds immediately to events emitted from another Lightning Component, but neither of them will break if the other is not present. This architecture is extremely flexible and powerful, since you could potentially have a large number of publishers and subscribers, all interacting with one another, but without tightly coupled relationships.

Just 2 years later, in 2017, a second event-driven architecture, platform events, was introduced to Salesforce developers. Whereas Lightning Events exist only in a user's browser, platform events are exchanged on the Salesforce platform itself and provide a way to exchange messages between standard Salesforce components,

custom components, and external systems. This is broadly referred to as the Salesforce Enterprise Messaging Platform (EMP), which hosts a common message bus on the Salesforce platform that systems can use to exchange messages.

Enterprise Service Buses (ESBs) have been popular for many years as an integration strategy between external systems. ESBs provide the same benefits described earlier by enabling external systems to have a loosely coupled relationship. The need for ESBs was driven by the rapid increase in complexity that comes from trying to integrate multiple systems that might each have different data formats. As shown in Figure 5-2, if you have to create direct integrations between n systems, you will have to create $n * (n - 1)$ direct integrations. If you have access to two-way "Point-to-point connectors" (such as a Salesforce to SAP connector), you will need $n * (n - 1)/2$ of these connectors. But if you use an ESB, you simply need n ESB connectors for each of n systems. The reduction in complexity is dramatic when you're connecting more than three systems. The same simplification holds true for the Enterprise Messaging Platform.

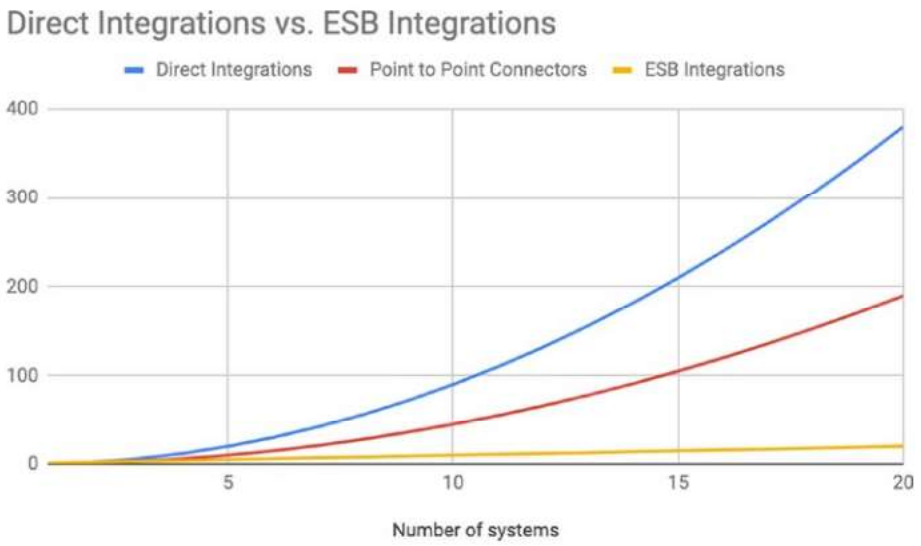

Figure 5-2. *Using an Enterprise Messaging Bus (ESB) requires far fewer integrations as system complexity increases*

Salesforce is increasingly emphasizing the use of the enterprise messaging platform, in part to address the need for data integration within its own sprawling portfolio of applications. Commerce Cloud, Marketing Cloud, and Salesforce's core platform have no native integration. Salesforce is beginning to use this event-driven architecture to ensure that these systems can keep their data in sync. A simple event like a user changing their

email address should ideally propagate across all of these systems. This is one reason why Salesforce is rolling out Change Data Capture capabilities to autogenerate events when important data is changed.

Using an event-driven architecture is a way to enable components in one package to communicate with components in another package. But dependency injection has benefits that make it more suitable in some circumstances. One of the benefits of Salesforce is that it performs business logic in transactions, so that if there's a problem, the entire transaction can be rolled back. Dependency injection allows you to take advantage of this so that components from across multiple packages can combine in a single transaction. It's now possible to delay sending platform events until after a transaction completes successfully, which prevents firing events prematurely for activities that might end up getting rolled back. But the publisher of an event and the subscriber to an event would still use two different transactions, thus making them perhaps *too loosely* coupled for some scenarios.

Enterprise Design Patterns

At Dreamforce 2012, Andrew Fawcett introduced the concept of Apex Enterprise Design Patterns, based on Martin Fowler's *Patterns of Enterprise Application Architecture*. That talk and his subsequent blog posts matured into the book *Force.com Enterprise Architecture*, soon to be in its third edition. The concepts introduced in that book have now been spread widely through a series of Trailhead modules that introduce Salesforce developers to the concept of Separation of Concerns and the Service,[11] Domain, and Selector layers.[12] You should get familiar with those Trailhead modules, and *Force.com Enterprise Architecture* is a very detailed and practical guide to implementing them.

Another of Andrew Fawcett's contributions is the FFLib Apex Commons[13] library on GitHub, which is designed to make these patterns easier to implement. This section contains just a brief summary of this topic as encouragement to consider using these patterns if you are dealing with a complex and growing codebase.

[11] https://trailhead.salesforce.com/content/learn/modules/apex_patterns_sl
[12] https://trailhead.salesforce.com/content/learn/modules/apex_patterns_dsl
[13] https://github.com/financialforcedev/fflib-apex-common

Separation of Concerns

The fundamental concept behind these enterprise design patterns is that you should have a clear separation of concerns when you write code. In general, there are four layers for you to consider:

- Presentation Layer—Provides the user interface
- Business Logic Layer—Services that manage calculations and other business logic
- Database Layer—Mechanisms to store data, the data model, and the stored data itself
- Data Access Layer—Selectors that allow you to retrieve from a database

This separation of concerns can be understood by considering different kinds of Salesforce metadata. Components like Page Layouts and the Lightning App Builder form part of the presentation layer, but don't allow you to customize business logic. Components like Validation Rules, Processes, and Workflow Rules allow business logic to be customized. The Salesforce object model itself, along with standard record edit pages, is part of the database layer. And Reports and Dashboards are part of the data access layer.

A common counterexample that does **not** employ separation of concerns is writing an Apex controller for Lightning or Visualforce that mixes together tools to manage the UI, handle calculations, update data, and access data. The reason this is an anti-pattern is that it hides business logic inside what seems to just be a controller for the user interface and prevents that logic from being reused if people access the data through a different channel such as the API. It also means that data access and updates are defined only in this one controller, even though other code in your org may require similar methods.

Separation of concerns allows for a clear logical distinction between parts of your system that handle these different jobs. Importantly, it also makes it easier to separate parts of your code across multiple packages, as described in the following sections.

Service Layer

> *The Service layer helps you form a clear and strict encapsulation of code implementing business tasks, calculations and processes. It's important to ensure that the Service layer is ready for use in different contexts, such as mobile applications, UI forms, rich web UIs, and numerous APIs. It must remain pure and abstract to endure the changing times and demands ahead of it.*[14]
>
> —Trailhead module "Learn Service Layer Principles"

As shown in Figure 5-3, the service layer centralizes business logic in a single place that can be accessed by many other types of component. Your business logic is one of the most critical customizations you can make on Salesforce. Calculations, logical conditions, and relationships between data can be business critical and sensitive to small mistakes. If that logic is distributed across many parts of your codebase, it makes it hard to understand and even harder to maintain. Where logic is defined using code, each group of logic should be held in a single service class.

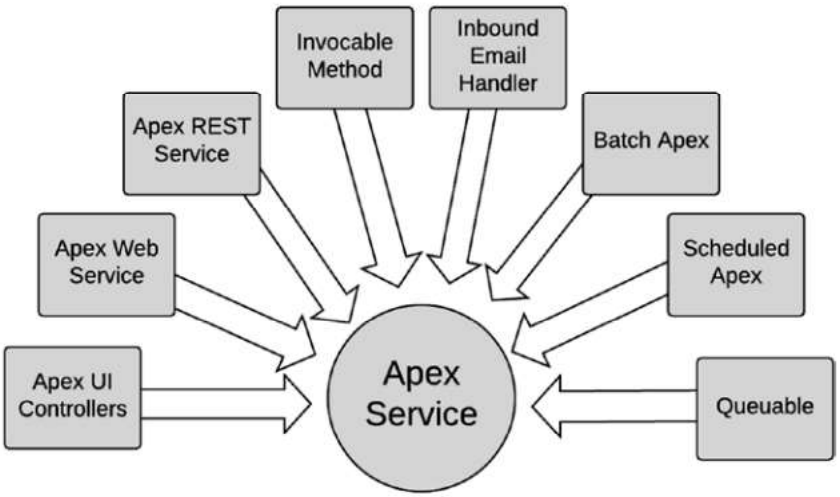

Figure 5-3. *The service layer provides services for many types of component*

For example, maybe you dynamically generate a commission on Opportunities that come from a channel partner. You should create a `PartnerCommissionService` that

[14] https://trailhead.salesforce.com/en/content/learn/modules/apex_patterns_sl/apex_patterns_sl_learn_sl_principles

centralizes the logic for that calculation. This service can then be used by Visualforce controllers, Lightning controllers, the API, Visual Flows, and so on. If you update the calculations, you only have to update that in a single place.

Implementing a service layer is generally simply a matter of copying code into a central place and updating references to point to that service class for their calculations. It's the first place for you to start if you want to implement these enterprise patterns, and it generally does not require much refactoring.

Unit of Work

The Unit of Work concept is a way to help manage database transactions and bulkification when implementing a service layer. As it says in the Trailhead module, "it's not a requirement for implementing a service layer, but it can help."

Whereas implementing service classes is generally quite simple, implementing the Unit of Work pattern, Domain Layer, and Selector Layer generally requires some specialized code to help them to work. This is where the FFLib Apex Commons package becomes particularly valuable, since it predefines interfaces and classes that can be used to implement this.

The basic idea of a unit of work is to allow a group of records to be passed from one class to another, being modified and updated as needed, before finally being committed to the database in a single transaction. The FFLib module enables you to manage parent-child relationships, bulkifies database operations, wraps multiple database updates in a single transaction to enable rollbacks, and more.

The reason this becomes relevant when building a modular architecture is that the Unit of Work object also provides a common data type for passing records between classes from multiple packages. The same is true with the following Domain and Selector layers: they provide an abstraction on top of DML and SOQL that allows multiple packages to collaborate on database transactions and queries.

These techniques add some overhead and complexity to your codebase, but reduce overhead and complexity as your project scales up. It's therefore important to become familiar with implementing these patterns so that you can make a more accurate cost-benefit assessment for each project. The Trailhead module on Separation of Concerns provides a chart[15] to help discern whether your project's scale justifies this approach.

[15] https://trailhead.salesforce.com/en/content/learn/modules/apex_patterns_sl/apex_patterns_sl_soc

Domain Layer

As mentioned, the Domain Layer is an abstraction on top of DML that allows you to manage validation, defaulting, and trigger handling logic in a single place that can be used by any class involved in updating data. The FFLib module provides interfaces and helpers for implementing a domain layer.

The basic idea is that there are some scenarios in which you need to apply a complex process consistently in both Triggers and in other DML updates. The domain layer provides this ability, as well as an object-oriented wrapper for dealing with native Salesforce objects. The FFLib naming convention is to name the Domain objects similarly to the native objects they represent. For example, you could use a Domain object called `Opportunities` that is a wrapper around the native Salesforce `Opportunity` object. This wrapper gives you access to additional methods and properties and allows you to attach logic (such as the `.applyDiscount` method shown in Figure 5-4) to the object when that's appropriate.

It's easy to misunderstand the Domain Layer as just another variation on trigger handlers. But it's actually much more than this. It provides a place to centralize business logic that is specific to a particular object, rather than allowing that logic to be distributed in ways that are difficult to understand or maintain.

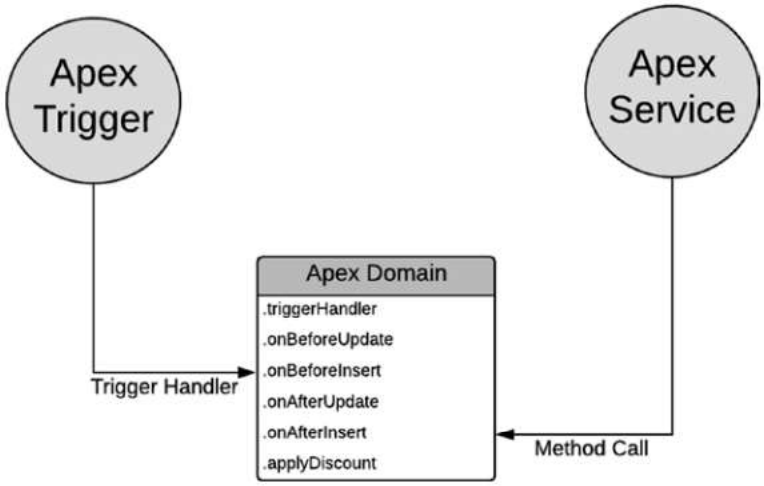

Figure 5-4. *The Domain Layer provides a wrapper around DML for both triggers and services*

Selector Layer

The Selector Layer is part of the data access layer used to retrieve data from a database. In Apex this means it's a wrapper around SOQL. Importantly, the Selector Layer provides an object-oriented wrapper around queries that allows for many helpful capabilities. When query definitions are distributed across many different classes, it becomes challenging to maintain them if fields change or new fields or filters become required. A Selector class addresses that by allowing you to specify the base query in a single place. It also allows queries to be passed between classes so that they can be modified if, for example, a class requires an additional field to be queried. Figure 5-5 shows how multiple classes can make use of a single Selector class.

Selectors allow you to use a "fluent syntax" in which multiple methods can be chained together to modify the query as shown in Listing 5-4. This becomes particularly powerful when passing Selectors across classes from different packages. Classes using the Selector can have confidence that all of the required fields have been selected and that appropriate row-level security has been enforced, and they can add requests for additional fields they may need to the same query before it is executed.

Listing 5-4. An example of Selector syntax showing multiple methods chained together (from Trailhead)

```
public List<OpportunityInfo> selectOpportunityInfo(Set<Id> idSet) {
  List<OpportunityInfo> opportunityInfos = new List<OpportunityInfo>();
  for(Opportunity opportunity : Database.query(
      newQueryFactory(false).
        selectField(Opportunity.Id).
        selectField(Opportunity.Amount).
        selectField(Opportunity.StageName).
        selectField('Account.Name').
        selectField('Account.AccountNumber').
        selectField('Account.Owner.Name').
        setCondition('id in :idSet').
        toSOQL()))
    opportunityInfos.add(new OpportunityInfo(opportunity));
  return opportunityInfos;
}
```

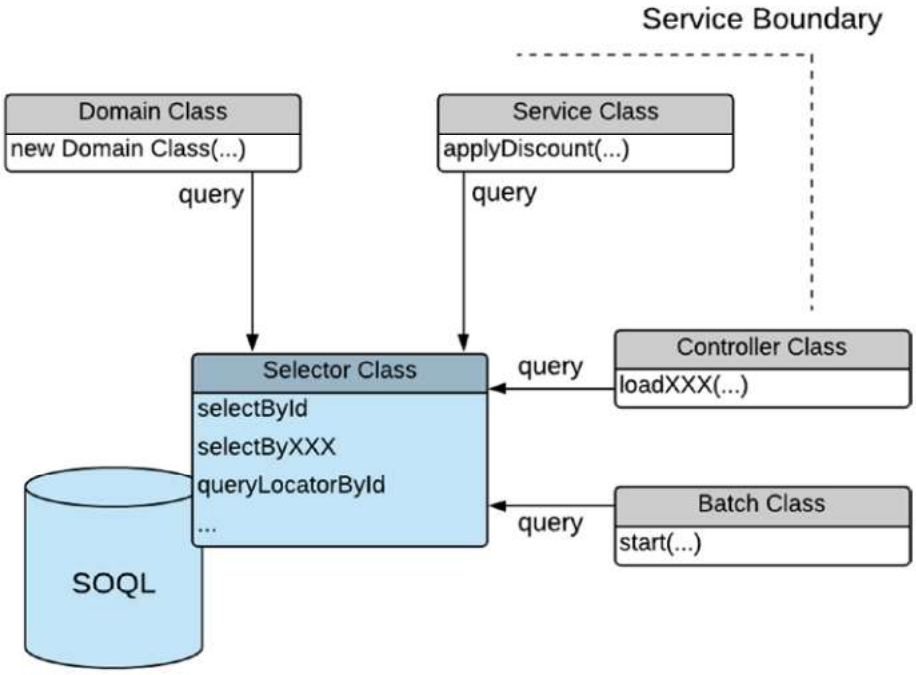

Figure 5-5. *The Selector Layer provides a common way to perform queries*

Factory Method Pattern

Implicit in the preceding patterns is the use of the Factory Method or "Application Factory" pattern. This is another example of Inversion of Control. The general idea is that instead of directly instantiating objects using, for example, new Opportunity(), you create methods that will return the object(s) you need. Listing 5-2 uses that approach when returning ShippingService. Adding this kind of indirection allows you to create mock interfaces for unit testing, polymorphic methods that determine their behavior at runtime, and allow you to gather repetitive boilerplate code in a single underlying method. The details of this are described briefly in a 2015 presentation by Andrew Fawcett.[16]

This is only a brief introduction to these topics. It is included here because of how helpful these patterns can be for dividing code across multiple packages. You should refer to Trailhead and Force.com *Enterprise Architecture* for a comprehensive guide.

[16] www.slideshare.net/andyinthecloud/building-strong-foundations-apex-enterprise-patterns/12

Trigger Management

Apex triggers allow for complex logic to be executed every time a database record is modified. As such, they're an extremely important part of the platform and are in widespread use across the Salesforce world. There are two main challenges related to Apex triggers, one old and one new. The old challenge has been a concern since I first encountered the language: how should you structure your triggers to make them maintainable and flexible? The general answer to that is to have one trigger per object and to delegate all the trigger logic to one or more trigger handlers. That advice worked well enough until we encountered the new challenge: if there should only be one trigger per object, how can we manage triggers if we need logic to be loosely coupled and distributed across multiple packages?

We'll first look at the classic "One Trigger Rule" before looking at strategies for handling triggers in a modular way.

The One Trigger Rule

Salesforce does not provide a way to determine the order in which triggers will execute. If you have more than one trigger on a single object, you can't predetermine the order in which those triggers will fire. When designing trigger logic, it's easy to encounter situations where you assume that processes happen in a particular order. But this can lead to very confusing scenarios in which a trigger seems to behave correctly sometimes but not other times.

It's also common to need to reference other data in your trigger using queries, and to iterate through records in a trigger, processing them one by one. Having more than one trigger can lead to redundant queries and redundant loops. All of these issues are addressed by having a single trigger for each object. Note that managed packages may supply their own triggers, but this isn't normally a concern.

The Trigger Handler Pattern

Triggers suffer from other limitations, such as the inability to define methods in them or to use most object-oriented design patterns. As a result, it's generally recommended to have little or no logic inside the trigger itself, but instead to create a trigger handler class that can hold all of this logic.