



Reinventing ITIL® in the Age of DevOps

Innovative Techniques to Make Processes
Agile and Relevant

Abhinav Krishna Kaiser



Apress®

Reinventing ITIL® in the Age of DevOps

**Innovative Techniques to Make
Processes Agile and Relevant**

Abhinav Krishna Kaiser

Apress®

Reinventing ITIL® in the Age of DevOps

Abhinav Krishna Kaiser
Bengaluru, India

ISBN-13 (pbk): 978-1-4842-3975-9

ISBN-13 (electronic): 978-1-4842-3976-6

<https://doi.org/10.1007/978-1-4842-3976-6>

Library of Congress Control Number: 2018965617

Copyright © 2018 by Abhinav Krishna Kaiser

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spaehr

Acquisitions Editor: Celestin Suresh John

Development Editor: Matthew Moodie

Coordinating Editor: Shrikant Vishwakarma

Cover designed by eStudioCalamar

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

ITIL® is a (registered) trademark of AXELOS Limited. All rights reserved.

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 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 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/978-1-4842-3975-9. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

To my parents,

Sandhya and Krishna Sharma,

without whom none of my successes would be possible.

Table of Contents

About the Author	xiii
About the Technical Reviewer	xv
Introduction	xvii
Chapter 1: Introduction to DevOps	1
What Exactly Is DevOps?.....	2
DevOps with an Example.....	3
Why DevOps?.....	4
Let's Look at the Scope	6
Benefits of Transforming into DevOps.....	7
Insight from State of DevOps Report.....	8
DevOps Principles	9
Culture	10
Automation	10
Lean.....	11
Measurement	12
Sharing	13
Elements of DevOps.....	13
People.....	16
Process.....	20
Technology.....	30
Is DevOps the End of Ops?.....	34

TABLE OF CONTENTS

Chapter 2: ITIL Basics.....	37
IT Service Management and ITIL.....	37
ITIL Conception	38
Competition to ITIL	40
Service Management in the Digital Age	41
Understanding Services	42
Service Types (Components)	43
Enabling Service.....	45
Enhancement Service.....	45
Understanding Processes	46
Understanding Functions	47
Functions in ITIL	47
Processes vs. Functions	48
ITIL Service Lifecycle	49
Service Strategy	50
Service Design.....	52
Service Transition	53
Service Operations	54
Continual Service Improvement	55
ITIL Roles.....	57
Service Owner	57
Process Owner	58
Process Manager.....	58
Process Practitioner	59
RACI Matrix.....	59
How Far Is ITIL from DevOps?.....	62
Chapter 3: ITIL and DevOps: An Analysis.....	63
Product vs. Services	64
Big-Ticket Conflicts.....	67
Which Is It: Sequential vs. Concurrent?	68
Let's Discuss Batch Sizes.....	68

TABLE OF CONTENTS

It's All About the Feedback	68
The Silo Culture	69
What Is Configuration Management?	70
Continuous Deployment Makes Release Management Irrelevant	71
Union of Mind-Sets	72
The Case for ITIL Adaptation with DevOps	73
To Conclude.....	74
Chapter 4: Integration: Alignment of Processes.....	77
Analysis of ITIL Phases	77
Analysis: Service Strategy Phase.....	79
Strategy Management for IT Services	79
Service Portfolio Management.....	82
Financial Management for IT Services	82
Demand Management	82
Business Relationship Management	84
Analysis: Service Design Phase	85
Design Coordination	85
Service Catalog Management	89
Service Level Management	90
Availability Management	91
Capacity Management.....	92
IT Service Continuity Management.....	95
Information Security Management.....	96
Supplier Management	99
Analysis: Service Transition Phase.....	100
Transition Planning and Support	100
Change Management	101
Service Asset and Configuration Management.....	101
Release and Deployment Management.....	101
Service Validation and Testing	101
Change Evaluation.....	102
Knowledge Management.....	103

TABLE OF CONTENTS

Analysis: Service Operation Phase.....	104
Event Management.....	104
Incident Management.....	106
Request Fulfillment	106
Problem Management	107
Access Management.....	107
Continual Service Improvement.....	107
The Seven-Step Improvement Process	108
Chapter 5: Teams and Structures	111
A Plunge into ITIL Functions.....	111
Service Desk.....	112
Technical Management	113
Application Management.....	115
IT Operations Management	116
DevOps Team Structure Revisited.....	118
Traditional Model	119
Agile Model.....	121
DevOps Model.....	125
Chapter 6: Managing Configurations in a DevOps Project.....	135
ITIL Service Asset and Configuration Management Process.....	135
Objectives and Principles	136
Service Assets and Configuration Items.....	136
Scope of Service Asset and Configuration Management.....	138
Introducing the CMDB, CMS, DML, and DS.....	138
Configuration Management Database	139
Configuration Management System	139
Definitive Media Library and Definitive Spares	140
Service Asset and Configuration Management Processes.....	142
Step 1: Management and Planning.....	143
Step 2: Configuration Identification	144

TABLE OF CONTENTS

Step 3: Configuration Control.....	144
Step 4: Status Accounting and Reporting.....	145
Step 5: Verification and Audit	146
Why Configuration Management Is Relevant to DevOps.....	147
Configuration Management in a DevOps Sense	147
Decoding IaaS.....	149
Decoding PaaS	149
Application Deployment and Configuration	150
Underlying Configuration Management.....	150
Automation in Configuration Management.....	151
Who Manages DevOps Configurations?.....	152
Comprehensive Configuration Management.....	153
Configuration Management Database	154
Source Code Repository	157
Artifact Repository.....	161
Management of Binaries	161
Chapter 7: Incident Management Adaption	163
What Is ITIL Incident Management?.....	164
Incident Management Is Vital	164
Incident Management Is the First Line of Defense.....	165
Digging Deeper into Incident Management.....	165
Objectives and Principles	165
Typical Process.....	168
Major Incidents.....	174
Incident Management in DevOps	175
Agile Project Management	176
Sprint Planning	179
Scope of DevOps Team in Incident Management	184
Knowledge at the Core	187
The DevOps Incident Management Process	191

TABLE OF CONTENTS

Chapter 8: Problem Management Adaption.....	201
Introduction to ITIL Problem Management.....	201
Objectives and Principles	202
Incidents vs. Problems	203
Key Terminologies in Problem Management	204
Problem Analysis Techniques.....	206
Brainstorming.....	206
Five-Why Technique	208
Ishikawa	211
Kepner-Tregoe	215
Typical Problem Management Process.....	216
Problem Management in DevOps.....	223
What Are the Possible Problems in a DevOps Project?	223
The Case for a Problem Manager	225
The DevOps Problem Management Process.....	226
Chapter 9: Managing Changes in a DevOps Project	231
What Constitutes a Change?.....	232
Overview of Resources and Capabilities	232
Change in Scope.....	233
Why Is Change Management Critical?	235
Objectives and Scope of ITIL Change Management.....	236
Types of Changes.....	237
Type 1: Normal Changes.....	238
Type 2: Emergency Changes.....	239
Type 3: Standard Changes.....	239
ITIL Change Management Process.....	240
Step 1: Create a Request for Change.....	242
Step 2: Assess and Evaluate the Change.....	243
Step 3: Authorize the Build and Test.....	243
Step 4: Build and Test.....	247
Step 5: Authorize the Implementation	247

TABLE OF CONTENTS

Step 6: Implement and Verify	247
Step 7: Review and Close the Change	247
How Are DevOps Changes Different from ITIL Changes?	248
The Perceived Problem with ITIL Change Management	249
Project Change Management	250
Risk Mitigation Strategies	254
DevOps Change Management Process	256
Change Management Adaption for Continuous Delivery	257
Change Management Adaption for Continuous Deployment	259
Maximum Agility with Standard Changes.....	262
Process for Identifying and Managing Standard Changes.....	264
Chapter 10: Release Management in DevOps.....	271
Change Management vs. Release Management.....	271
Release Management vs. Release and Deployment Management	273
Basics of a Release.....	274
Release Unit	274
Release Package	275
Types of Release.....	276
Early Life Support.....	277
Deployment Options	278
Four Phases of Release Management.....	280
Release and Deployment Planning	280
Release Build and Test	281
Deployment	281
Review and Close	282
Releases in DevOps	282
Sequential and Iterative Nature of the Process	282
Release Management Process Adaption with Iterations	284
Expectations from Release Management.....	285
The Scope of Release Management.....	287
Automation of Release Management	288

TABLE OF CONTENTS

The DevOps Release Management Team	289
Release Management Team Structure.....	290
Welcome, Release Manager, the Role for All Seasons	291
Product Owners Are the New Release Managers	294
Index.....	297

About the Author



Abhinav Krishna Kaiser works as a senior manager in a leading consulting firm. He consults with organizations in the areas of Agile, DevOps, and ITIL and leads programs involving the development and implementation of Agile and DevOps solutions.

He is one of the leading names synonymous with ITIL on the Web, and his previous publication, *Become ITIL Foundation Certified in 7 Days*, is one of the top guides recommended to IT professionals looking to get into the service management field and become ITIL Foundation certified.

Abhinav started consulting with clients 15 years ago on IT service management, creating value by developing robust service management solutions. He is one of the foremost authorities in the area of configuration management, and his solutions have stood the test of time, rigor, and technological advancements.

Abhinav has trained thousands of IT professionals on DevOps processes, Agile methodologies, and ITIL expert-level certifications. He blogs and writes guides and articles on DevOps, Agile, and ITIL at <http://abhinavpmp.com>. His first book came out in 2015: *Workshop in a Box: Communication Skills for IT Professionals*.

While the life of a consultant is to go where the client wants him to be, Abhinav is currently in Bangalore. He is happily married to Radhika, and they have two children Anagha, daughter and Aadwik, son.

About the Technical Reviewer



Kwok Tsang is a certified professional IT architect, ITIL expert, and accredited trainer with a proven track record in IT service management, Agile DevOps, and enterprise and solution architecture design. He has held senior positions in IT consulting firms and played key roles for customer projects in the APAC region. He has worked in multicultural DevOps teams that focus on ITIL v3 service management systems and processes, project management, business analysis, Lean IT, enterprise architecture (TOGAF), IT governance (COBIT 5), strategy and road map, systems integration, information security, enterprise solutions cloud computing, and software-defined networks. He is responsible for end-to-end solution architecture and design, including presales consulting, proposal development, delivery, knowledge transfer, and project completion. He has worked on medium to large-scale projects including cross-country systems and has designed IT and network solutions for large corporations globally.

Introduction

Reinventing ITIL® in the Age of DevOps, my third publication, came more out of necessity rather than an idea that sprang into my mind one fine morning. My clients often ask me about the relevance of ITIL today with the advent of DevOps. They question on Yammer and other social networks whether the need to implement and maintain ITIL is made redundant in DevOps projects. After a few discussions over Skype and drinks, they are convinced that the world still needs ITIL, even if the DevOps clouds are to take over. Thus started the journey of this book with my intent to reach out to a wider community, not only convincing them of ITIL's relevance but also showing them how ITIL can work in DevOps projects.

In this book, I have not shied away from getting into the details of ITIL and DevOps and mapping activities between the two. I have provided my views on all 26 ITIL processes and 5 ITIL functions and how they fit into the DevOps scheme of things. For a few major processes such as incident and change management, I have delved so deep that one can pick up my book and start implementing with minor situational adaptations.

I don't expect you to be ITIL aware or DevOps aware or Agile aware. As long as you are in the IT field and understand the concepts around software development and support, you will definitely be able to understand all the concepts that are discussed in book.

Specifically, Chapter 1 is DevOps 101 for those who are new to DevOps. Even if you are well versed in DevOps, I still encourage you to read this chapter as the understanding of DevOps varies slightly from one person to another, and there are no formal DevOps methodologies defined. While I connect the dots in the later chapters with the DevOps concepts, you will be in a position to understand the mapping from the word go. Chapter 2 is ITIL 101 where I introduce various ITIL concepts, the service lifecycle, processes, and other ITIL disciplines. If you are ITIL Foundation certified, you can safely skip this chapter. However, I have been witness to various misconceptions around ITIL concepts. So, it would be prudent to skim over the details of this chapter in the interest of the rest of the book.

INTRODUCTION

Chapters 3 and 4 are dedicated to integration and alignment of DevOps processes with ITIL phases and processes. Yes, you read that right. Every single ITIL process (26 in total) has been analyzed from the perspective of DevOps. The various ITIL roles and DevOps roles that exist, the team structures, the synergies, and the irrelevancies are discussed in Chapter 5. Chapters 6 through 10 are dedicated to the major ITIL processes that play a significant role in DevOps projects: configuration management, incident management, problem management, change management, and release management.

The ideas presented in *Reinventing ITIL® in the Age of DevOps* are one way of combining the ITIL framework with DevOps practices. There are a number of other possible ways. Therefore, I want to hear from you, discuss more, and possibly collaborate. Please share your thoughts, feedback, and suggestions at <http://abhinavpmp.com>.

CHAPTER 1

Introduction to DevOps

New ways of working or new methodologies begin to unearth because of a problem--yes, it all starts with a problem. DevOps too had its own reasons. Businesses craved for fast turnarounds of their solutions. And often businesses found out in the midst of development that they didn't have all the information they needed to make the right decisions. They wanted to recommend a few more changes to the requirements and still expected the delivery to happen on time. DevOps was born to solve this problem.

Well, DevOps just didn't show up as the DevOps we have today. It has evolved over time. It was clear to those who started solving the problem around agility that DevOps has a lot more potential to not just solve the problem of agility but also increase productivity by leaps and bounds. Further, the quality of the software developed had the potential to be at a historic best. Thus, to this day, DevOps keeps changing and changing for the better.

DevOps is not just a methodology for developers. Operations too has its share of the benefits pie. With increased automation, operations went from being a mundane job to an innovative one. Operations folks just got themselves a new lease of life through various tools that can make their working lives a whole lot fun, and they could start looking forward to integrating and configuring tools to do advanced stuff, rather than the repetitive workload that's generally associated with operations. Here too, the productivity shot up, and human errors became a rarity.

The software development was carried out on the back of the software delivery lifecycle (SDLC) and managed through waterfall project management. On the operations front, ITIL ruled the roost. Through DevOps, development and operations essentially came together to form a union. In the mix, the waterfall methodology gave way to Agile methodologies, and still people who design DevOps processes did not have a good understanding of how ITIL would come into DevOps. So, a lot of noise started to circulate that the dawn of DevOps is the end for ITIL. This is plainly noise without any substance; you will find out through the rest of the book the value that ITIL brings to the table and why DevOps cannot exist in its entirety without a framework such as ITIL.

The book is structured around the ITIL service management framework and will explore what changes need to be made to ITIL if it needs to ease into DevOps projects. I have covered every single ITIL process and ITIL function with respect to DevOps in Chapter 4, and Chapters 5 through 10 provide in-depth analysis of major processes in ITIL around DevOps designs and implementations. You can use the contents of this book readily to start implementing ITIL in the most effective manner for it to create value in DevOps projects.

In this chapter, I briefly explain DevOps, including its principles, elements, and processes. Chapter 2 provides a snapshot of ITIL, including its lifecycle, phases, processes, and functions. I analyze DevOps and ITIL in Chapter 3, identifying the commonalities and conflicts, to support in the journey toward adapting ITIL for DevOps implementations.

What Exactly Is DevOps?

There are multiple perceptions about DevOps in the core. In fact, if you search the Web, you will be surprised to find multiple definitions for DevOps and that no two original definitions converge on common aspects and elements.

I have trained thousands in the area of DevOps, and the best answer I have is that it brings together the development and operations teams, and that's about it. Does bringing two teams together create such a strong buzz across the globe? In fact, if it actually was just the culmination of two teams, then probably DevOps would have been talked of in the human resources ecosphere, and it would have remained a semicomplex HR management process.

During the beginning of the DevOps era, to amuse my curiosity, I spoke to a number of people to understand what DevOps is. Most bent toward automation, some spoke of *that* thing they do in startups, and there were very few who spoke of it as a cultural change. Interesting! Who talks of culture these days, when the edge of our seats burn a hole if we don't act on our commitments? A particular example made me sit up and start joining the DevOps dots, and it all made sense eventually.

DevOps with an Example

Let's say that you are a project manager for an Internet banking product. The past weekend you deployed a change to update a critical component of the system after weeks of development and testing. The change was deployed successfully; however, during the post-implementation review it threw out an error that forced you to roll back the change.

The rollback was successful, and all the artifacts pertaining to the release were brought to the table to examine and identify the root cause the following Monday. Now what happens? The root cause is identified, a developer is pressed into action to fix the bug, and the code goes through the scrutiny of various tests, including the tests that were not originally included that could have caught the bug in the functional testing stage rather than in production. All tests run OK, a new change is planned, it gets approved by the change advisory board, and the change gets implemented, gets tested, and is given all green lights.

These are the typical process activities that are undertaken when a deployment fails and has to be replanned. However, the moment things go south, what is the first thing that comes to your mind as the project manager? Is it what objective action you should take next, or do you start thinking of the developer who worked in this area, the person responsible for the bug in the first place? Or do you think about the tester who identified the scenarios, wrote the scripts, and performed exploratory testing? Yes, it is true that you start to think about the people responsible for the mess. Why? It is because of our culture. We live in a culture that blames people and tries to pass the buck.

I mentioned earlier about some respondents telling me that DevOps is about culture. So, what culture am I talking about with the context of this example? The example depicts a blameful culture where the project manager is trying to pin the blame on the people within his team directly responsible for the failure. He could be factually right in pinning the blame on people directly responsible, but I am focusing on the practice involving blaming individuals.

How is this practice different from a DevOps culture? In DevOps, the responsibility of completing a task is not considered as an individual responsibility but rather a shared responsibility. Although an individual works on a task, if the person fails or succeeds, the entire team gets the carrot or the stick. Individuals are not made responsible when

we look at the overall DevOps scheme of things, and we don't blame individuals. We follow a blameless culture. This culture of blamelessness culminates from the fact that we all make mistakes because we are humans after all and far from being perfect. We make mistakes. So, what's the point in blaming people? In fact, we expect that people do make mistakes, not based on negligence but from the experimentation mind-set. This acceptance (of developers making mistakes) has led us to develop a system where the mistakes that are made get identified and rectified in the developmental stages and much before they reach production.

How is this system (to catch mistakes) built? To make it happen, we brought the development and operations teams together (to avoid disconnect), we developed processes that are far more effective and efficient than what is out there (discussed in the rest of the book), and finally we took umbrage under automation to efficiently provide us with feedback on how we are doing (as speed is one of the main objectives we intend to achieve).

DevOps is a common phrase, and with its spread reaching far and wide, there are multiple definitions coming from various quarters. No two definitions will be alike, but they will have a common theme: culture. So, for me, DevOps is a cultural transformation that brings together people from across disciplines to work under a single umbrella to collaborate and work as one unit with an open mind and to remove inefficiencies.

Note Blameless culture does not mean that the individuals who make repeated mistakes go scot-free. Individuals will be appraised justly and appropriately but in a constructive manner.

Why DevOps?

What gave rise to a new culture called DevOps, you might ask. The answer is evolution. If you take a timeline view of software, from the 1960s up to the advent of the Internet in 1990s, developing software was equivalent to a building project or launching a space shuttle. It required meticulous planning and activities that were planned to be executed sequentially. The waterfall project management methodology was thus born with five sequential steps, as indicated in Figure 1-1.

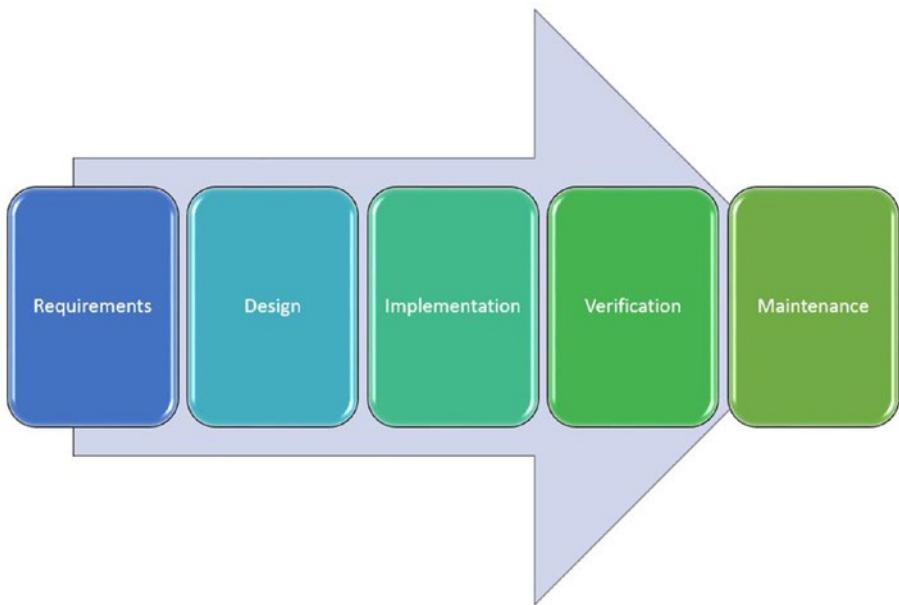


Figure 1-1. Waterfall project management methodology

When the Internet boomed, the software was far more accessible than the earlier era, and this generated demand not seen earlier. When the software industry started to expand, the waterfall model's limitations were exposed. The need to complete a detailed planning exercise and the sequential practice of flow did seem like an impediment to the advancement of the software industry.

Then in 2001 at a ski resort in Utah, the Agile Manifesto was born. A number of prevalent Agile methodologies came together to form a common goal that would remove the cast-in-stone waterfall sequential activities.

Agile was more fluid because they did not conceive of all of the requirements at the beginning and try to solve world hunger overnight. It was an approach that was based on iterations, where all the activities of project management just cycled over and over again. In between, if a requirement was to change, that's okay because there were provisions to make changes that were not bureaucratic nor tedious in nature. In fact, the Agile methodology puts the emphasis on the response to changes in requirements rather than a map to be followed.

The flexibility and dynamism that came about through Agile spread its wings across the software industry. A number of software projects migrated to the Agile way of working, and to this day, there are projects that are undergoing serious coaching during this transformational phase.

The Agile methodology is pretty simple where you keep things small enough to manage and large enough to be rendered meaningful. The time frames that defined iterations in Agile did not carry too much wriggle room. From an efficiency perspective, Agile was far better than the waterfall model. However, the demands from the market were out of sync with what Agile could provide. While the market shouted out for faster deliveries, the need to increase quality (reducing defect rate) was perennially being pursued. The Agile project management methodology needed something, like an elixir to run things faster. It needed automation. Enter DevOps!

Automation by itself is like giving a drone to a kid without really teaching him the process to make it fly. Generally speaking, technology by itself has no meaning if there are no underlying functional architecture, process, and embedded principles. DevOps, therefore, is not just automation but a whole lot more. You will find out the nitty-gritty details in the coming sections.

Let's Look at the Scope

The word *DevOps* gives away the scope through its conjunction of two parts of a software lifecycle. While Agile existed mainly to put an end to the rigidity brought forth by the waterfall model, it was said that the methodology can be used for operations as well. But, without an overarching process or a framework, using Agile for operations with the same rigor was not going to work. DevOps bridged this gap by bringing in the operational phases along with the developmental activities under a single umbrella and applying common processes and principles to be employed across the board.

DevOps comes into play when you get started with the software development process, which is the requirement gathering phase. It ends when the software retires from service. DevOps spans the entire wavelength of a software lifecycle, and if you read between the lines, you cannot just implement and execute DevOps until deployment and be done with it. It will continue to function until the software is used by its designated users. In other words, DevOps is here to stay, and stay for as long as services are delivered. So, in practice, the operational phase runs perpetually, and DevOps will deliver the required optimization and automation. The processes to run operations will be borrowed from the ITIL service management framework, and the present format of the ITIL framework will be highly customized to fit the DevOps bill. It is this specific ITIL fit into a DevOps project that is at the heart of this book.

Note The word DevOps came into existence thanks to Twitter. The first Devopsdays conference was held in Ghent, Belgium, in 2009. While people tweeted about it, the #devopsdays tag ate way 11 characters out of a possible 140. In a way of shortening it, one of the tweeters used #devops, and others followed suit. This led to the advent of what we know today as DevOps.

Benefits of Transforming into DevOps

A number of software companies have been delivering applications for a number of years now. Why do we need DevOps to tell us how we must develop now?

Our services are being delivered to a number of customers including top banks and mines around the globe. I am running just fine with my service management framework. Why DevOps?

People have lived for thousands of years now. They did just fine, reproducing and surviving. What has changed in the past 100 years? We have changed the modes of transport for better efficiency, we communicate faster today, and overall our quality of life has gone up several notches. *Something is working* is not a barrier for improvements to be brought about and to transform. DevOps introduces several enhancements in the areas of working culture, process, technology, and organization structure. The transformation has been rooted in practices that were developed by some like-minded organizations that were willing to experiment, and the results have vastly gone in the favor of DevOps over other ancient methodologies that still exist today.

Amazon, Netflix, Etsy, and Facebook are some of the organizations that have taken their software deliveries to a whole new level, and they don't compete anymore with the laggards. They have set new benchmarks that are impossible to meet with any of the other methodologies.

At the 2011 Velocity conference, Amazon's director of platform analysis, Jon Jenkins, provided a brief insight into Amazon's ways of working. He supported it with the following statistics:

During weekdays, Amazon is able to deploy every 11.6 seconds on average. Most organizations struggle to deploy weekly consistently, but Amazon does more than 1,000 deployments every hour (1,079 deployments to be precise). Further, 10,000 hosts receive

deployments simultaneously on average, and the highest Amazon has been able to achieve is 30,000 hosts simultaneously receiving deployments. Wow! These numbers are really *out of this world*. And these are the statistics from May 2011. Imagine what they are able to do today!

It's just not the speed of deployments. There are several added advantages that Amazon went on to claim during the conference.

- Outages owing to software deployments have reduced by a whopping 75 percent since 2006. Most outages are caused by new changes (read software deployments), and the reduction in outages points to the success achieved in deploying software changes.
- The downtime owing to software deployments too has reduced drastically, by about 90 percent.
- On an average, there has been an outage for every 1,000 software deployments, which is about a 0.001 percent failure rate. This looks great for a moderate software delivery organization, but for Amazon, the number seems high because of the 1000+ deployments every hour.
- Through automation, Amazon has introduced automatic failovers whenever hosts go down.
- Architecture complexity has reduced significantly.

Insight from State of DevOps Report

Puppet Labs publishes an annual whitepaper called the “State of DevOps Report.” The report provides insight into the world of DevOps, the statistics, the changes brought about, and all the new innovations that have come about in the past year.

In the 2017 report, Puppet Labs surveyed about 3,200 professionals including executives, developers, testers, and other IT professionals. It is abundantly clear from the report that the number of people working in DevOps has gone up steeply. In 2014, 16 percent of the respondents worked in DevOps, and in 2017, it was 27 percent. DevOps is indeed quickly reaching the far-cornered IT companies, and I wouldn't be surprised if the jumps are exponential in the coming years.

- *Insights from development:* IT companies that have implemented DevOps are able to deploy 46 times faster than non-DevOps organizations. Their change lead time is a dramatic 440 times faster as well.
- *Insight from operations:* Organizations practicing DevOps have been able to recover from failures 96 times faster than their non-DevOps peers and have reduced the change failures by 80 percent.
- *Employee attraction:* It was reported that potential employees find it more appealing to work in organizations that follow DevOps methodologies than otherwise.

DevOps Principles

DevOps principles or the guidance toward the true north is in a state of constant evolution. In fact, there are multiple versions of the principles. The most widely believed set of principles is represented with the acronym CALMS. Figure 1-2 represents a mug from a marketing campaign for DevOps featuring CALMS.



Figure 1-2. DevOps principles (credit: devopsnet.com)

CALMS stands for the following:

- Culture
- Automation
- Lean
- Measurement
- Sharing

Culture

There is a popular urban legend that the late Peter Drucker, known as the founder of modern management, famously said, “Culture eats strategy for breakfast.” If you want to make a massive mind-boggling Earth-shaking change, start by changing the culture that can make it happen and adapt to the proposed new way of working. Culture is something that cannot be changed by a swift switching process. It is embedded into human behavior and requires an overhaul of people’s behavior.

These are some of the behavioral traits that we want to change with DevOps:

- Take responsibility for the entire product and not just the work that you perform
- Step out of your comfort zone and innovate
- Experiment as much as you want; there’s a safety net to catch you if you fall
- Communicate, collaborate, and develop affinity with the involved teams
- For developers especially, you build it, you run it

Automation

Automation is a key component in the DevOps methodology. It is a massive enabler for faster delivery and also crucial for providing rapid feedback. Under the culture principle, I talked about a safety net with respect to experimentation. This safety net is made possible through automation.

The objective is to automate whatever possible in the software delivery lifecycle. The kinds of activities that can be efficiently automated are those that are repetitive and those that don't ask for human intelligence. For example, building infrastructure was a major task that involved hardware architects and administrators, and most importantly building servers took a significant amount of time. This was time that was added to the overall software delivery. Thanks to the advancement of technology, we have cloud infrastructure today, and servers can be spun up through code. Additionally, we don't need hardware administrators to do it. Developers can do it themselves. Wait, there's more! Once the environment provisioning script is written, it can be used to automate spinning up servers as many times as necessary. Automation has really changed the way we see infrastructure.

Activities involving executing tasks such as running a build or running a test script can be automated. But, the activities that involve human cognizance are hard to automate today. The art of writing the code or test scripts require the use of human intelligence, and the machines of today are not in a position to do it. Tomorrow, artificial intelligence can be a threat to the activities that are dependent on humans today.

Lean

DevOps has borrowed heavily from the Lean methodology and Toyota Production Systems (TPS). The thinking behind the Lean methodology is to keep things simple and not to overcomplicate them. It is natural that the advent of automation can decrease the complexity of architecture and simplify complicated workflows. The Lean principle aids in keeping us on the ground so we can continue working with things that are easy to comprehend and simple to work with.

There are two parts to the Lean principle. The primary one is not to bloat the logic or the way we do things; keep it straightforward and minimal. An example is the use of microservices, which support the cause by not overcomplicating the architecture. We are no longer looking to build monolithic architectures that are cumbersome when it comes to enhancements, maintenance, and upgrades. A microservice architecture solves all the problems that we faced yesterday with monolithic architectures; it is easy to upgrade, troubleshoot (maintain), and enhance.

The second part of the principle is to reduce the wastage arising from the methodology. Defects are one of the key wastes. Defects are a nuisance. They delay the overall delivery, and the amount of effort that goes into fixing them is just sheer waste of time and money. The next type of waste focuses on the convoluted processes. If something can be done by passing the ball from A to B, why does it have to bounce off C? There are many such wastes that can be addressed to make the software delivery more efficient and effective.

Measurement

If you ought to automate everything, then we possibly need a system to provide feedback whenever something goes wrong. Feedback is possible if you know what the optimum results are and what aren't. The only way we can find out whether the outcome is optimum or not is by measuring it. So, it is key that you measure everything if you are going to automate everything!

Measurement principle provides direction on the measures to implement and the tabs to keep to feel the pulse of the overall software delivery. It is not a simple task to measure everything. Many times, we don't even know what we should measure. Even if we do it, the *how* part can be an obstacle. A good DevOps process architect can help solve this problem. For example, if you are running static analysis on your code, the extent of passable code must be predetermined. It is not a random number, but a scientific reasoning must be behind it. A number of companies allow a unit test to pass even if it parses 90 percent of the code. We know that ideally it must be 100 percent, so why should anybody compromise for 90 percent? That's the kind of logic that must go behind measuring everything and enabling fast feedback to be realistic about the kind of feedback that you want to receive.

In operations, monitoring applications, infrastructure, performance, and other parameters come under this principle. Measurements in monitoring will imply when an event will be categorized as a warning or an exception. With automation in place, it is extremely important that all the critical activities, and the infrastructure that supports them, be monitored and optimized for measurement.

There are other measurements as well that are attached to contracts and SLAs and are used for reporting on a regular basis. These measurements are key as well in the overall scheme of things.

Sharing

The final principle is sharing, which hinges on the need for collaboration and knowledge sharing between people. If we aim to significantly hasten the process of software delivery, it is only possible if people don't work in silos anymore. The knowledge, experience, thoughts, and ideas must be put out into the open for others to join in the process of making them better, enhanced, and profound.

One of the key takeaways of this principle is to put everyone who works on a product or a service onto a single team and promote knowledge sharing. This will lead to collaboration rather than competition and skepticism.

There are a number of collaboration tools on the market today that help support the cause. People don't even have to co-located to share and collaborate. Tools such as Microsoft Teams and Slack help in getting the information across not only to a single person but to all those who matter (such as the entire team). With information being transparent, there will be no reason for others to worry or be skeptical about the dependencies or the outcome of the process.

Elements of DevOps

DevOps is not a framework; it's a set of good practices. It got started out of a perfect storm that pooled several practices together (which will be discussed later in this chapter), and today we consider them under the DevOps umbrella. You might have seen the elephant in Figure 1-3. The IT industry around software development is so vast that a number of practices are followed across the board. This is depicted as the elephant in the figure. DevOps, which is a cultural change, can be applied to any part of the software industry and to any activity that is being carried out today. So, you can identify any part of the elephant (say testing) and design DevOps-related practices and implement them and you are doing DevOps!

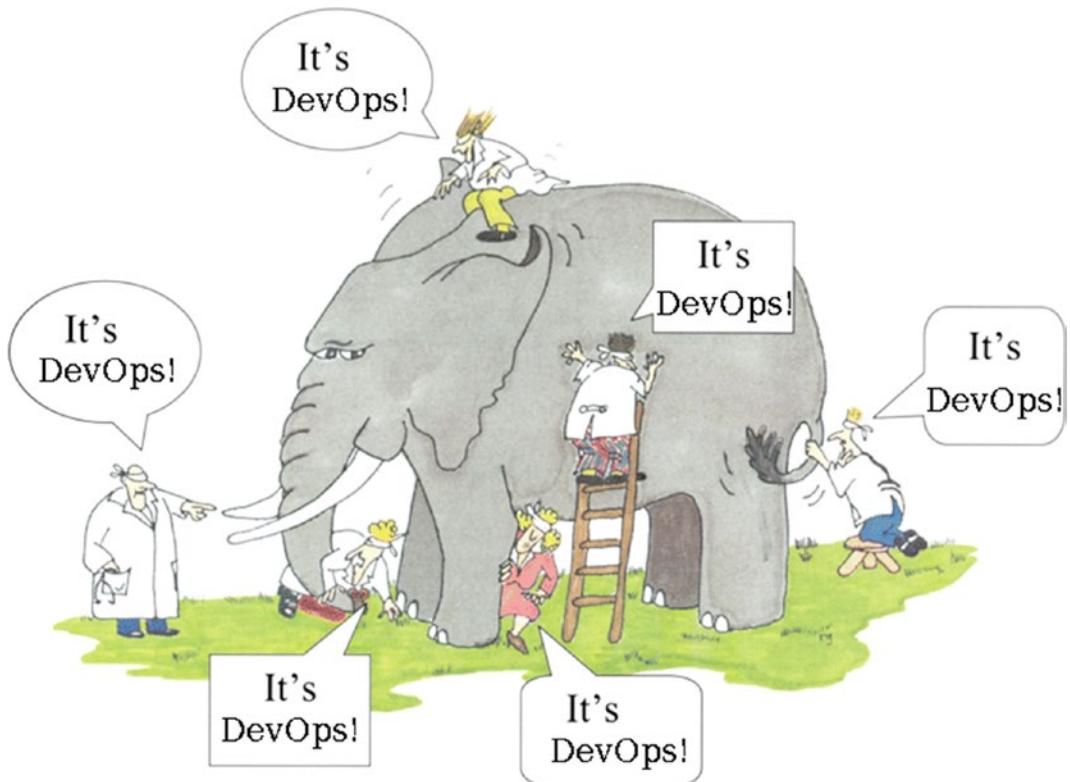


Figure 1-3. *The DevOps elephant (credit: devopsdays.org)*

No matter where you want to implement DevOps, there are three common elements that support and enable the culture change. The three sections are indicated by a Venn diagram in Figure 1-4.

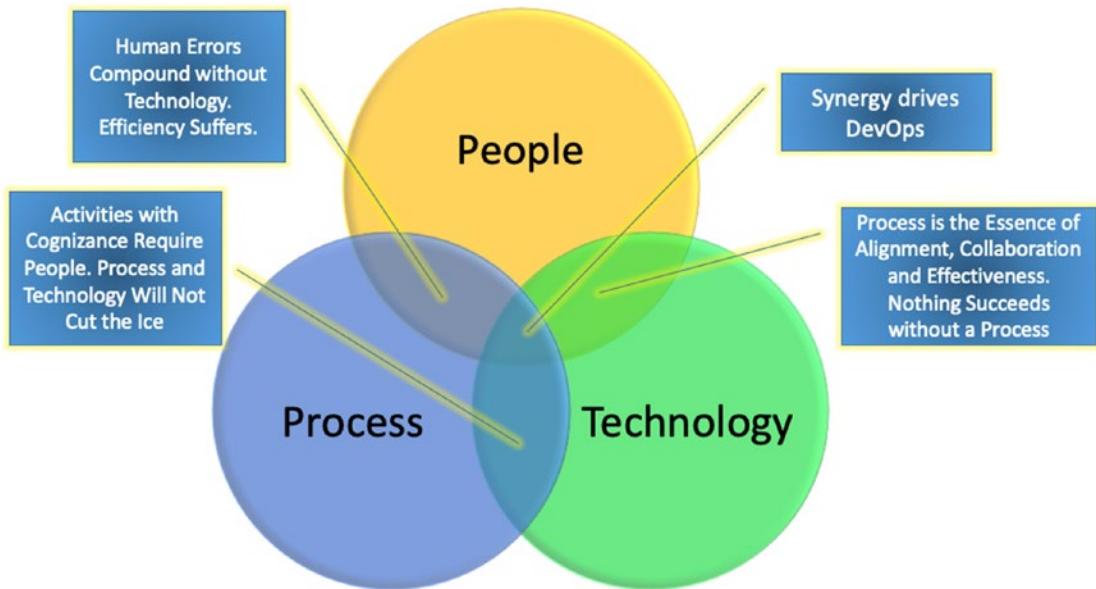


Figure 1-4. Three elements of DevOps

People, process, and technology are the three elements that are common to all DevOps practices. In fact, they are the enablers to effect change in the DevOps culture. Only when the three elements come together in unison are we able to realize the complete benefits of DevOps.

Let's examine the three elements and see how they fit together. To bring in a cultural change, we most definitely need people, and people cannot operate without the aid of processes. By bringing in people and processes, we have achieved the functional design to implement a DevOps solution. However, the question to ask is whether it is efficient. Humans are known to make mistakes. We cannot avoid it. How can processes alone support the humans into identifying the mistakes committed? There may be a way to do, but it is most definitely not efficient. To make things move faster and in an efficient manner, we need the technology stack to help us achieve the process objectives.

Today people talk of DevOps through the lens of technology. They throw around several tool names and claim that they do DevOps. So, the question to ponder is whether you can really do DevOps by tools alone. Can people and technology elements suffice without an underlying process? You probably guessed the answer, and the answer is no. Nothing succeeds without a process in place, not only in DevOps but in every objective that you want to achieve, IT or otherwise.

This is the age of artificial intelligence. Some experts claim that the machines will take over the world and replace the work people used to do. There are a number of movies such as *Terminator Genisys* and *Ex Machina* that play to the tunes of AI taking over the reins and putting humans in jeopardy. I love fiction, and AI making decisions is something I like to think of fiction (for now anyway because the technology advancements are breaking new barriers every day). However, coming back to DevOps, just employing technology with underlying processes is not going to cut it. Without people, creation does not happen. Yes, technology can automate a number of preprogrammed activities, but can it develop new solutions on its own? I don't think so, not today anyway. People are the driving force of creation and the agents of cultural change.

All the three elements of people, process, and technology are essential to build the DevOps methodology and to achieve the objectives that are set forth before us. By the union of all three elements, we can create an unmatched synergy that can fuel developments at an unparalleled pace.

People

The word *DevOps* is derived from the conjunction of two words, development and operations. I have already familiarized you with what DevOps is all about: a change of culture in the way we deliver and operate the software. People are at the heart of this cultural transformation, and they are one of the three critical elements that enable the DevOps culture.

The development and operation teams are amalgamated to bring about a change in culture. The thinking behind it is quite straightforward. Let's say that an application is developed and it comes to the change advisory board (CAB) for approval. One of the parties on the CAB is the operational teams. They specifically ask questions around the testing that has been performed for this software, and even though the answer from development is yes for the success rate of all the tests, the operational teams tend to be critical. They don't want to be blockers, yet they find themselves in a position where they have to support software that they haven't been familiarized yet. The bugs and defects that come with the software will become their problem after the warranty period (usually between one and three months). Most important, they only have the confirmation of the developers to go with when the quality of the software is put on the line.

In the same scenario, imagine if the operational teams were already part of the same team as development. Being on the same team will give them an opportunity to become familiar with the development process and the quality controls put in place. Instead of asking questions in the CAB, they can work progressively with the development teams in ensuring that the software is maintainable and all possible operational aspects are undertaken beforehand. This is one such case study that showcases the benefit of having a single team. I will talk more about it under the section “DevOps Team.”

In Figure 1-5, you can visualize the development team on one end of the cliff, while the operations team is on the opposite. In between the two cliffs lies an area of uncertainty where activities that fall between the two teams have a knack for being unpredictable and they are usually sparred over, generally over ownership. In other words, you want things to be either with the development team or with the operations team. There is no bridge between the teams, meaning there can be a lot of confusion, miscommunication, and mistrust between the two opposing teams.



Figure 1-5. Conflict between development and operations teams

Let's play out the priorities for both teams. The development team still has a job because there is a need to develop new features. That is their core area, and that is what they must do to remain relevant. The operations team's big-ticket goal is to

keep the environment stable, in the most basic sense. They need to ensure that even if something was to go wrong, they are tasked to bring it back to normal, in other words, maintain the status quo. So, here we have a development team intending to create new features and an operations team looking to keep the environment stable. Why does it have to be rocket science to have evolved into a methodology called DevOps that promises to shake the industry from its roots? Well, the environment is going to remain stable if there are no changes introduced to it. As long as it stays stagnant, nothing ever will bother its stability, and the operations team would have been awarded for a stellar job. But, we have the development team waiting in the wings to develop new features. New features that are developed will be deployed in the production environment. And there is every chance that the deployment of new features could impact stability. So, stability is something that can never be achieved as long as new features are introduced, and no software will remain stagnant without enhancements and expansion.

A decent way to tackle this conundrum between the development and operation teams is to put them together and to create channels of communication within the team members. Both the development and operations teams have a shared responsibility to ensure that development, testing, deployment, and other support activities happen smoothly and without glitches. Every team member takes responsibility for all the activities being carried out, which translates to the development and operation teams jointly working on the solution that begins with coding and ends with deployment. The operation teams will have no reason to mistrust the test results and can confidently deploy onto the production environment.

DevOps Team

Along with a new way of working and the new culture comes what is called a *DevOps team*. We have so far discussed a single team consisting of the development and operation teams. An anti-pattern that has emerged in the IT industry is the creation of DevOps teams that consist of a pool of tooling professionals. This team consisting of tool specialists is not the true DevOps team in essence; rather, it should be a truly cross-functional team consisting of roles needed to support an application.

Basis for a DevOps Team

DevOps does not suggest that you pool your entire set of development and operation teams together and create a DevOps team. The DevOps team must be built around an application. If application X is being developed, let all the people responsible for its development and responsible for its operations be brought together to create one single team, which is the true DevOps team. If application X is complex with a number of features, find a way to logically create multiple DevOps teams based on the application's features.

An Example of a DevOps Team

Application X is an Internet banking program that caters to individuals and small business owners. It is currently in the development stages. Let's think of the roles that are required to support it. Today most projects work in an Agile manner, and the development of application X will be no different. It is based on Scrum practices and employs a single Scrum team for its development. The DevOps team for application X possibly consists of the following roles:

Product owner (PO): The product owner is from the business organization and is the owner of the product backlog.

Scrum master (SM): The Scrum master leads the development as a servant leader.

Developer (DEV): The coding bit and unit testing are carried out by the developers.

Testers (TEST): Testers are involved in developing test scripts and executing functional and nonfunctional tests.

Architect (ARC): Architects design the software and are generally shared across multiple DevOps teams as they are not required to play a full-time role in a single DevOps team.

Database administrator (DBA): This person does database management.

Application support (AS): This person is responsible for the support activities of the application.

System administrator (SYS): This person is responsible for configuring and managing tools.

Service manager (SMG): This person is responsible for managing services from the incident, problem, change, and other service management areas.

IT security (SEC): This person is responsible for managing aspects of IT security.

Figure 1-6 provides a typical structure of a DevOps team in which the architect is shared between the two illustrated DevOps teams.

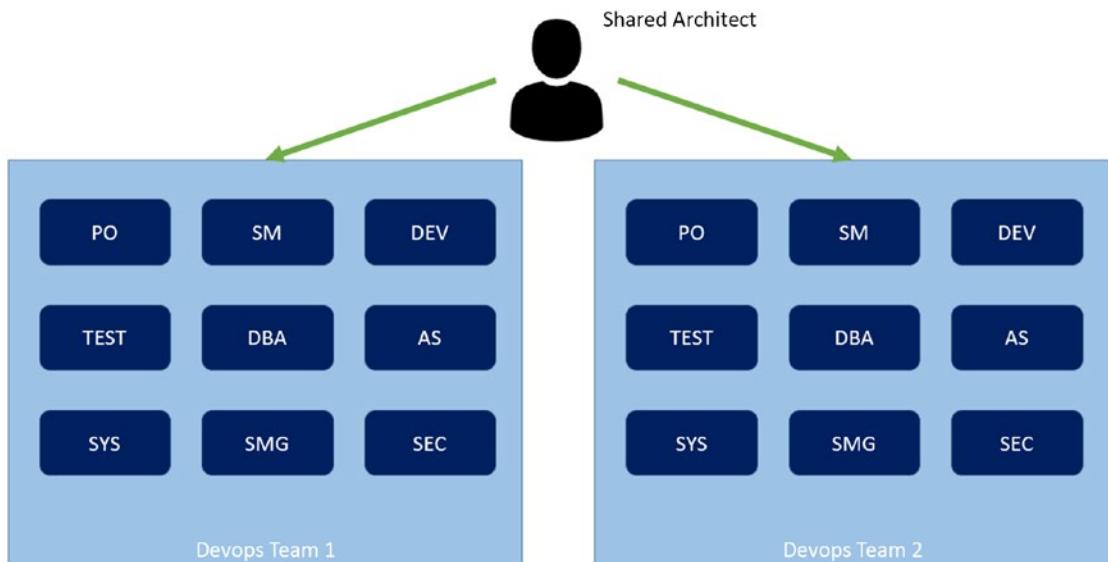


Figure 1-6. Typical DevOps team structure

Process

Processes are a key component in ensuring the success of any project. However, we often find that most DevOps implementations focus more on automation and technology and give a backseat to processes that are supposed to be the basis for automation. They say that backseat driving is dangerous, so placing processes in this position and hoping that the destination would be reached in record time with no mishaps is a gamble that plays with unpredictability. Therefore, it is important that processes are defined first along with a functional DevOps architecture and then translated into tooling and automation. *The process must always drive tools and never the other way around.*

With DevOps combining different disciplines under a single banner, the processes too need to be rejigged to fit the new objectives. In this section, I will cover the processes pertaining to the development area. The rest of the book is dedicated to operational processes and their union with the development processes.

Waterfall project management methodology (such as PMI-backed Project Management and PRINCE for projects in controlled environments) are not favored in the IT field anymore. There are various reasons going against this methodology, mainly ebbing from the rigidity it brings into the project management structure. Most IT projects are run on Agile project management methodologies because of the flexibility it offers in this ever-changing market. According to PMI's Pulse of Profession publication, 71 percent of organizations have been leveraging Agile. Another study by PricewaterhouseCoopers in a study named "Agile Project Delivery Confidence" report that Agile projects are 28 percent more successful than their waterfall counterparts. This is huge considering that Agile is still new and emerging and the waterfall methodology has existed since the 1960s.

When we talk about Agile project management, there are a number of methodologies to pick from. Scrum, Kanban, Scrumban, Extreme Programming (XP), Dynamic Systems Development Method (DSDM), Crystal, and Feature Driven Development (FDD) are some examples. However, all the methodologies are aligned by a manifesto that was formulated in a ski resort in Utah in 2001. And there are a set of 12 Agile principles that provide guidance in setting up the project management processes.

In this book, I will not go into the Agile project management processes. These processes are similar irrespective of DevOps implementation. The specific DevOps processes that are introduced on top of the Agile processes are as follows:

- Continuous integration
- Continuous delivery
- Continuous deployment

Continuous Integration

A number of developers work together on the same piece of code, which is referred to as the mainline in the software development lingo. When multiple developers are at work, conflicts arising due to changes performed on pieces of code and the employed logic are quite common. Software developers generally integrate their pieces of code into the mainline once a day.

When conflicts arise, they discuss and sort it out. This process here of integrating the code manually at a defined time slows down the development. Conflicts at times can have drastic results with hundreds of lines of code having to be rewritten. Imagine the time and effort lost due to the manual integration. If I can integrate code in almost real time with the rest of the developers, the potential amount of rework can significantly be reduced. This is the concept of *continuous integration*.

To be more specific, continuous integration is a process where developers integrate their code into the source code repository (mainline) on a regular basis, say multiple times a day. When the code is integrated with the mainline, any conflicts, if there are any, will come out into the open, as soon as it is integrated. The resolution of conflicts does not have to be an affair where all developers sit across the codebase and break their heads. Only those who have conflicts need to sort them out manually. By doing this conflict resolution multiple times a day, the extent of conflicts is drastically minimized.

Note The best definition of continuous integration was coined by Martin Fowler from ThoughtWorks, who is also one of the founding members of the Agile Manifesto.

Continuous integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily, leading to multiple integrations per day. Each integration is verified by an automated build (including tests) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly (source: <https://www.martinfowler.com/articles/continuousIntegration.html>).

Integrating the code with the mainline is just the beginning. Whenever the code is integrated, the entire mainline is built, and other quality checks such as unit testing and code-quality checks (static and dynamic analysis) also are carried out.

Note *Build* is a process where the human-readable code is converted into the machine-readable language (executable code), and the output of a build activity is a binary.

Unit testing is a quality check where the smallest testable parts of an application are tested individually and in a componentized manner.

Static analysis is an examination of the source code against the coding standards set forth by the industry/software company, such as naming conventions, blank spaces, and comments.

Dynamic analysis is an examination of the binary during runtime. Such an examination will help identify runtime errors such as memory leaks.

An Illustration

Let's say a particular project has three developers, and each developer integrates their code three times a day. On a daily basis, this equates to nine integrations every day. As per Figure 1-7, code that is integrated gets unit tested first, followed by software build and code quality checks. All this happens automatically whenever code gets integrated.

With nine integrations on a daily basis, we are staring at a possibility of having nine unit tests, nine builds on the entire mainline, and nine code quality checks.

Suppose one of the builds or unit tests or code quality checks fail. The flow gets interrupted, and the developer gets down to work to fix the defect at the earliest. This ensures that the flow of code does not get hampered and other coders can continue coding and integrate their work onto the mainline.

In the "Technology" section, I will talk about a few tools that are used to achieve this kind of an automation that will set loose the dependencies that we normally have and the impediments that are normally faced by developers.

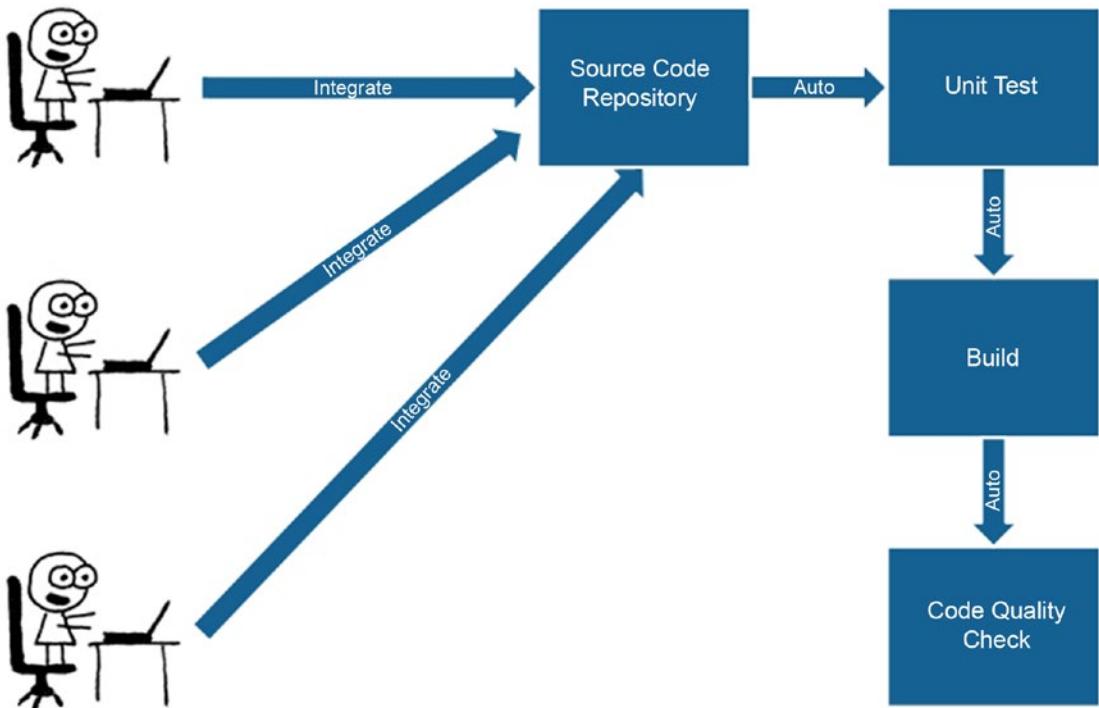


Figure 1-7. Continuous integration

Continuous integration allows for fast delivery of software, and any roadblocks are avoided or identified as early as possible, thanks to rapid feedback and automation. The objective of the continuous integration is to hasten the coding process and to generate a binary without integration bugs.

Continuous Delivery

With continuous integration, we have achieved these two things:

- A binary is generated successfully.
- Code-level and runtime checks and analysis are completed.

The next item in the software delivery lifecycle (SDLC) is to test the generated binary from various angles, aspects, and perspectives. Yes, I am referring to the tests such as system tests, integration tests, regression tests, user acceptance tests, performance tests, and security tests; this list is quite endless. When we are done with the agreed number of tests, the binary is deemed to be of quality and deployable into production. The qualified

binary can be deployed into production with a click of a button. The qualification of any of the binaries as releasable into production is termed as *continuous delivery* (see Figure 1-8). It is generally seen as a natural extension of the continuous integration process.

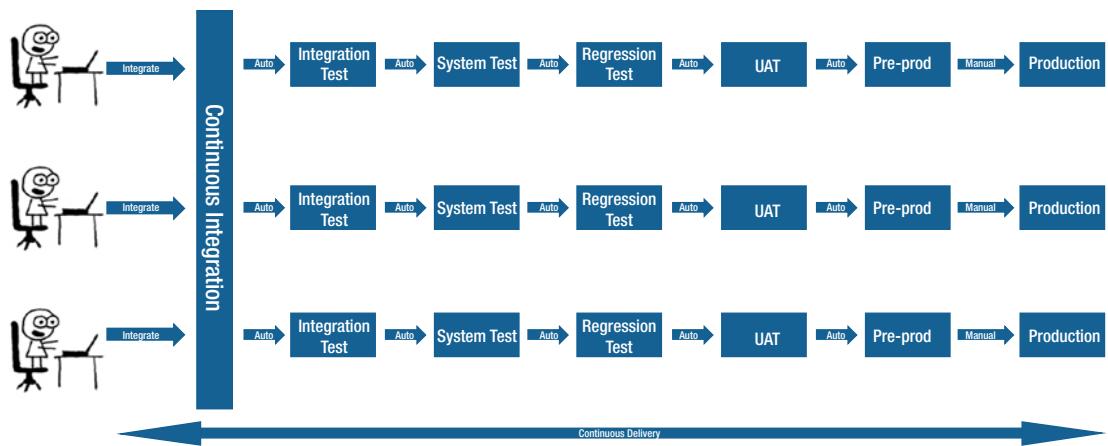


Figure 1-8. Continuous delivery

Figure 1-8 depicts a continuous delivery pipeline. After every successful cycle of continuous integration, the binary is automatically subjected to an integration test. When the integration test is successful, the same binary is automatically system tested. The cycle passes on up to the preproduction environment as long as the tests (regression and user acceptance testing [UAT] in this illustration) are successful. When the same binary is successfully deployed in the preproduction environment (in this illustration) or any other environment that comes before the production environment, the binary becomes qualified to be deployed on to production. The deployment into the production environment is not done automatically but requires a trigger to make it happen. The entire cycle starting from the code push into the source code repository up to the manual deployment into the production environment is continuous delivery.

In the illustration, I have shown three developers integrating their code and three deployable binaries. Continuous delivery does not dictate that all three binaries have to be deployed into production. The release management process can make a decision to deploy only the latest binary every week. Remember that the latest binary will consist of the code changes performed by all the developers up until that point in time.

The sequence of automation for the activities beginning in the continuous integration process up until the production environment is referred to as a *pipeline* or *continuous delivery pipeline* in this case.

Who Employs Continuous Delivery?

Continuous delivery gives an organization complete control over the production environment. Only the binaries that have passed manual scrutiny (such as change management) will pass through the gate between the preproduction and production environments.

DevOps implementation is generally done in a step-wise manner. Organizations first play around with a few DevOps tools trying to automate little pieces of work and then the complete activities. The first major step toward DevOpsification (yes, I just coined this word) is the definition and implementation of continuous integration. After gaining confidence in doing it, the next round will be to integrate and automate tests without the need for any trigger (automation). Combining the test integrations into the pipeline where the binary can be deemed deployable is the big step toward achieving continuous delivery. Most organizations do not have the capability to fully implement continuous delivery as illustrated in Figure 1-8. It requires adequate focus, unparalleled talent in process and technology, and most importantly a concrete vision and intent to move toward automation.

Automation Testing vs. Continuous Testing

Here is more DevOps jargon for you: continuous testing. *Continuous testing* is the process where automated tests kick in after the continuous integration process. There is absolutely no human involvement during the execution of tests, not even a trigger to begin a test. Everything happens in a sequence, and the only automated trigger is a successful test of the previous activity. For example, in Figure 1-9, the automated UAT does not happen if the regression test does not succeed.

The term automation testing may be familiar to you. What is the difference between the two?

Automation testing is a process in which the execution of tests is done automatically using various automation testing tools. However, the trigger to begin the testing in an automated fashion is a manual activity.

Continuous testing is a process where the execution of tests is run through the testing tools automatically following code integration and success of the previous sequential activity. There is no manual trigger to begin a test execution.

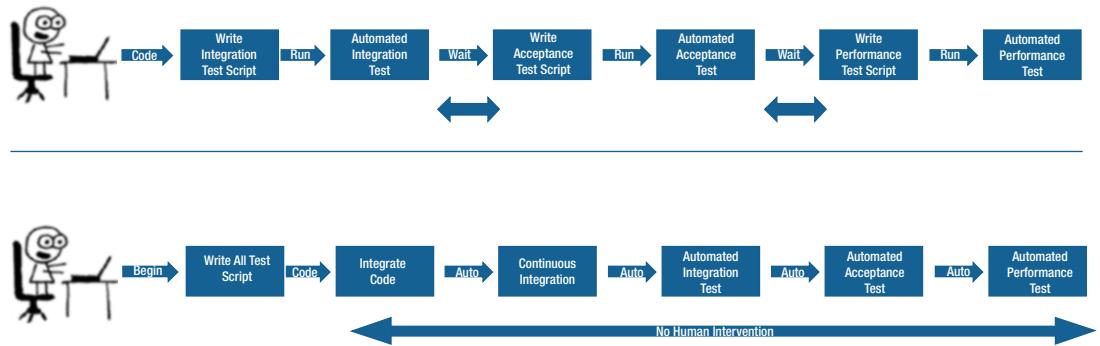


Figure 1-9. The difference between automation and continuous testing

So, what is the difference between automation testing and continuous testing?

In both the cases, test scripts have to be written by testers, and there is no difference in the manner they are written. The only difference is the execution.

Generally, in a project involving automation testing, the code is first developed and built. Then the automation test script is written and fed into the testing tool and manually triggers an automated test.

In the continuous testing world, the automation test scripts are written before the coding begins, which requires a good and common understanding of the requirements. When the binary is successfully built and continuous integration cycle is successfully processed, the execution of the tests is triggered and run automatically as well.

The advantage of continuous testing over automation testing is that the entire sequence of activities in the pipeline happens rapidly, one after another. There are no waiting periods for scripts to be ready or for the tester to hit the Execute button. It is swift, leaving no room for inefficiencies of human constructs and therefore enabling faster delivery, which is the objective of DevOps.

Continuous Deployment

Continuous deployment is one step beyond continuous delivery. In continuous delivery, the deployment to production is based on a manual trigger. However, in the *continuous deployment* process, the deployment to production happens automatically, as depicted in Figure 1-10.

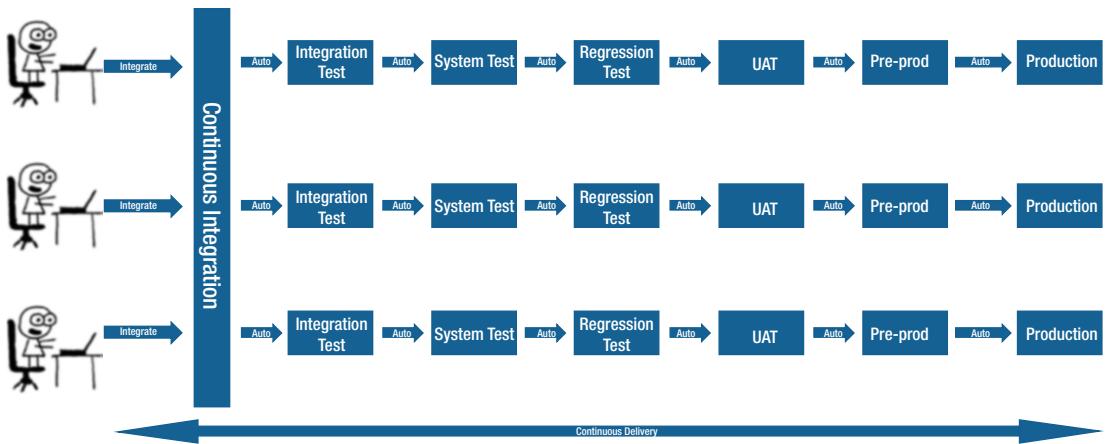


Figure 1-10. Continuous deployment

In Figure 1-10, as soon as all the tests are successful, the binary is deployed to the preproduction environment. When the deployment to preproduction goes as planned, the same binary is deployed into production directly. In continuous delivery (Figure 1-8), the binaries were qualified as deployable, and the release manager was in a position to not deploy every single qualified binary into production. On the contrary, in continuous deployment, every single qualified binary gets deployed onto the production instance.

You might think that this is far too risky. How can you deploy something into production without any checks and balances and approvals from all stakeholders? Well, every single test that is performed and all the quality checks executed are the checks that are qualifying binaries as deployables. It's all happening in an automated fashion. You would do the same set of things otherwise but manually. Instead of deploying multiple times a day, you might deploy once a week. All the benefits that you derive from going early into the market are missing from the manual processes.

Let's say that one of the deployments were to fail. No problem! There is an automated rollback mechanism built into the system that rolls back the deployment within seconds. And it is important to note that the changes that are being discussed here are tiny changes. So, the chances of these binaries bringing down a system are remote.

Continuous Delivery vs. Continuous Deployment

Figure 1-11 depicts the difference between continuous delivery and continuous deployment.

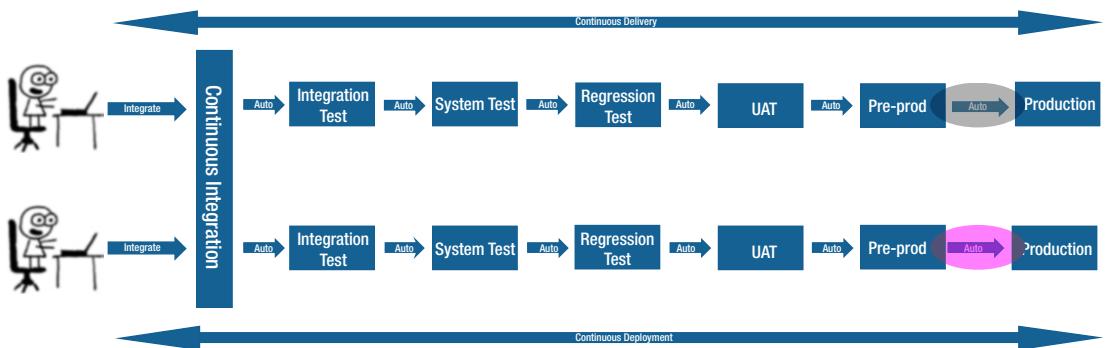


Figure 1-11. Difference between continuous delivery and continuous deployment

The difference lies in the final sequence, where the deployment to production instance is automatic in continuous deployment and has a manual trigger for continuous delivery.

Any organization on a journey of implementing DevOps will implement the continuous delivery process and upon gaining sufficient maturity will move toward the pinnacle of DevOps maturity: the continuous deployment process.

Organizations that feel a need to keep total control of their production environment through a formal structure of approvals and visibility tend to opt for continuous delivery. Banking and other financial segments fall into this category.

There are other organizations that have scaled the DevOps maturity ladder and are quite confident that the automatic deployment doesn't cause significant impact to their production environment, and even if something was to fail, then the rollback will be rapid too, even before anybody can notice it. Companies like Amazon, Netflix, and Google have been in this space for a while now. I shared a statistic earlier about Amazon managing a deployment every 11.6 seconds. How is it even possible? Look no further than continuous deployment.

Note Here's a cheat sheet for continuous delivery and continuous deployment:

Continuous delivery: You can deploy.

Continuous deployment: You will deploy.

Technology

Technology is the third element of DevOps and is often regarded as the most important. It is true in a sense that without automation, we cannot possibly achieve the fast results that I have shared earlier through some statistics. It is also true that technology on its own, without the proper synchrony of people (roles) and processes, is like a spaceship in the hands of kindergarteners. It is a must that the people and process sides of DevOps be sorted out first before heading this way.

The number of tools that claim to support DevOps activities is enormous, too many to count.

Choosing the Right Tool

Not all tools can be used for all technologies. And the same tool cannot be used for carrying out different types of activities. For example, if I am using the Java programming language for unit testing, I need to opt for a tool such as JUnit. On the other hand, for Microsoft technologies, for unit testing I need NUnit.

However, there are tools that support multiprogramming languages such as Jenkins and Cucumber. In fact, for the same activity and the same technology, there are multiple tool choices. It is important to weigh in on the capabilities and compatibilities before choosing one. For example, for the source code repository function, there are multiple good choices. Git is the most common one, and Subversion is almost as popular as Git. Both provide versioning capability for source code and can be integrated with other toolsets for automation. Which one should we choose? If we look deeper into the technology employed, Subversion falls under the category of central version control system (CVCS), and Git is under distributed version control system (DVCS).

The underlying technology specifies where the code is stored and retrieved. CVCS employs a server-client relationship to store and retrieve the code. The developer is required to first check out the existing code, make changes, and check it back in. Under DVCS, every single developer has the entire mainline sitting on their computer. So, there is no concept of check-out and check-in. DVCS allows multiple developers to work seamlessly.

The greatest advantage of DVCS over CVCS is its availability. For DVCS to function, you don't need an active network connection, but for CVCS it is absolutely necessary. If server access is blocked, development comes to a complete standstill. Thus, CVCS has a single point of failure (SPOF), something that goes against the principles of DevOps,

which emphasizes incessant development and maximum efficiency. Since source code is available locally in DVCS, accessing, merging, and pushing the code is much faster relative to CVCS. DVCS also enables software developers to exchange code with other developers before pushing it to the central server and, consequently, to all other developers. In fact, there are no notable disadvantages with DVCS other than perhaps the storage needs on a developer's terminal if source code contains an elongated history of changesets.

In this example of a source code repository, I gave a glimpse of how the tools are chosen and how they must be scrutinized. The role of a DevOps architect is to find the right tool for a particular activity.

Categories of Tools

Remember the periodic table of elements? XebiaLabs has replaced the chemical elements with (a select few) DevOps tools in their respective categories in a periodic table format. Figure 1-12 is the periodic table of DevOps at the time of writing.

Figure 1-12. Periodic table of DevOps (Source: <https://xebialabs.com/periodic-table-of-devops-tools/>)

If I have to categorize the toolsets, then I would probably do it based on its function and the outcome that I am trying to achieve. For example, managing the source code is an outcome, so I will categorize all source code repositories under a single bucket. The following section are some categories to help you identify the right tool. This, by no means, is comprehensive.

Source Code Repositories

These are source code repositories:

- *Git*: The most popular source code repository today. Git is free and open source. However, some Git hosting providers charge you for the hosting service and the customizations that have been employed.
- *Apache Subversion*: Commonly referred to as SVN and is open source (and free). This version control system is losing steam because of the underlying CVCS technology. This is not apt for most DevOps implementations.
- *Mercurial*: Another DVCS tool that is playing catch-up with Git.

Hosting Services

These are hosting services:

- *Amazon Web Services (AWS)*: AWS has taken the DevOps world by storm. It has transformed the way hosting is done, where consumers pay only for what they use. The AWS ecosystem is vast, and Amazon offers its own set of tools for carrying out various DevOps activities. For example, AWS CodeCommit is an AWS instance of the Git software.
- *Azure*: Microsoft is not far behind with its offerings and is on an equal footing with AWS. It hosts multiple platforms and is not limited to the Windows operating system.
- *Google Compute Engine*: Can you leave out Google when talking about hosting and state-of-the-art tooling solutions? The solutions offered on all hosting services are of high quality, so in most cases, the cost becomes a factor in choosing one over the other. At the time of this writing, Google Compute Engine offers the cheapest prices across multiple segments.

Orchestrators

Automation is achieved through the orchestration of various toolsets. Orchestrators offer the ability to create workflows through which pipelines can be defined along with various parameters that set the criteria for progression into the next sequential activity. These are orchestrators:

- *Jenkins*: Jenkins is the most popular orchestrator as it works across platforms and across technologies. It can talk to most of the tools through various plug-ins. The best part is that the tool is free and comes with massive community support.
- *UrbanCode Deploy*: UrbanCode Deploy is an IBM product that is a powerful orchestration tool. Its compatibility with mainframe systems is its biggest advantage. The tool can also carry out automated deployments, which Jenkins achieves through orchestration with other tools.
- *Bamboo*: Bamboo is from Atlassian and not free. Like Jenkins, it works across platforms and supports multiple technologies.

Deployment and Environment Provisioning

These do deployment and environment provisioning:

- *Ansible*: Ansible is a popular tool for automated deployments, environment provisioning, and configuration management. The concept of infrastructure as code (IAC) where infrastructure can be built by scripts is handled through Ansible. Other toolsets in this category that are discussed also effectively manage IAC.
- *Puppet*: Puppet comes in two versions: open source with limited features and full-blown enterprise version. It is perhaps the most popular tool in this category.
- *Chef*: This is similar to Ansible and Puppet. Facebook uses Chef for its deployments and managing the configuration of applications and infrastructure.

Testing

There are a number of types of testing, and in this section, I will highlight the popular testing tools that are employed in DevOps implementations.

- *Selenium*: This is the most popular functional testing tool. It works across platforms and technologies and integrates seamlessly with all the major orchestration tools. It is open source.
- *Cucumber*: Cucumber is employed for running automated acceptance tests (such as UAT without the user) . It provides support for behavior-driven development (BDD), which is a development methodology that is driven by writing the test script first and then the code. The test scripts are written in a natural language called Gherkin.
- *HP LoadRunner*: LoadRunner by HP is a testing tool for measuring system behavior and performance under load. Performance testing is considered as nonfunctional testing where the quality aspects of an application are scrutinized.

Is DevOps the End of Ops?

With the introduction of continuous integration, continuous delivery, and continuous deployment, the focus has been to plug the defects, increase the quality, and not sacrifice the efficiency. The thinking behind the notion that of DevOps ending the operational activities is based on the premise that lack of defects will not give rise to operational work. If there are no defects, there are potentially no incidents or problems, which translates to a massive reduction in operational work. Another example is if we implement continuous deployment, the change and release management processes as we know them will be automated to a great extent and will diminish the need for approvals and subsequent approvals, release planning, and release deployment.

Let's get one thing straight: no matter how much we try with the use of technology and automation, defects will always exist. The number of defects will come down due to the rapid feedback and automation, but to state that all the defects would be identified and rectified is absurd. With the reduction of defects, the amount of operational work will definitely go down. With the argument around change and release management processes, the execution of changes and releases can be automated through continuous

delivery and continuous deployment, but the planning bit will always remain the cognizance of human experience. To an extent, the operational work involving change and release management processes are starting to go down as well.

Innovation is a double-edged sword. With the introduction of tools and automation, there is a new operational requirement to set up and configure the tool stack and to maintain it as long as the project is underway. This is an operational activity that is added to the traditional operations work. While some areas have seen a reduction, there are new ones that have sprouted to take their place. The manual, repetitive, and boring activities are going away. In their place, exciting DevOps tooling work has come to the fore and is making the operational roles all the more lucrative.

So, if you are an operational person, it is time to scale up and scale beyond managerial activities alone. The new wave of role mapping requires people to be technomanagerial in talent and be multiskilled. T-shaped resources, not only for operations but also in development, are being sought after. I-shaped resources must look toward getting acquainted with areas of expertise that complements their line of work.

With the advent of DevOps, there is a turbulence created in software projects. This is a good turbulence because it seeks to raise the level of delivery and to make teams, rather than individuals, accountable for the outcomes. From an operations front, it is clear that their role has gone up a couple of notches where the mundane, boring, repetitive activities have been replaced with imaginative and challenges jobs such as configuring pipelines, integrating toolsets, and automating configuration management. The nature of work for operations has changed but not the role they play as guardians of environments and troubleshooters of incidents and problems. DevOps has not spelt the end of operations but rather rejuvenated it to an exciting journey that will keep the wits of people working in operations alive.

CHAPTER 2

ITIL Basics

The Information Technology Infrastructure Library (ITIL) is the most popular framework that exists today to deliver services. It has become a standard of sorts, and most service-based organizations have implemented one form of ITIL or another.

The objective of the ITIL framework is to provide guidance on how services have to be defined, developed, built, and operated. The framework provides a detailed lifecycle of phases from inception to operation in a methodical fashion, which some construe as loaded or heavy (something not looked too favorably today). No matter what the critics say, ITIL is complete and absolute and takes into account all perspectives of a service; it is a valuable ally for a service management organization. The entire process of setting up ITIL in organizations may take anywhere from 6 to 18 months depending on the volume and complexity.

I have practiced ITIL for more than 15 years, and when I look at the length and breadth of the framework, it amazes me how holistically it has grown over the years. I even penned a book on the subject: *Become ITIL Foundation Certified in 7 Days* (Apress, 2016). This book is a foundational course in ITIL for those who intend to get into the service management industry. The additional aim of the book is to aid you in becoming ITIL Foundation certified (within seven days considering professionals have a day job on hand). If you want to get a deeper understanding of the ITIL framework, I highly recommend that you read that book. This chapter is meant to provide the absolute basics of ITIL, which is the foundation for the building that I am about to construct in the rest of the book.

IT Service Management and ITIL

There was a time when there was business and then there was IT. Businesses had their set of practices, and IT was a supporting agent, helping businesses achieve their tasks, such as supplying the business with a word processor for drafting contracts and providing them with the ability to compute complex formulas. Without IT, businesses could survive, although surely with some inconvenience.

Today, the world of business has been turned on its head. You take IT out of business and the business will cease to exist. In other words, there is no business without IT. The business relies on IT for its sustenance, and IT is not a support function anymore. Rather, it is a partner that enables businesses to achieve their goals and succeed in beating their competitors. Try to think of a midsize business where IT may not be involved. Aha! I know your results came up blank. To reiterate, IT is a part of the business, and there is no looking back.

IT service management is defined as the implementation and management of quality IT services that meet the needs of and deliver value to the business. IT services are provided by IT service providers (the entity that provides IT services to internal and external customers) through an appropriate mix of people, processes, and information technology.

There is increased pressure on IT to deliver on its services. IT must deliver services that not only meet its objectives but also do it effectively and efficiently. And it must be done at a minimal cost. The competition in the IT service management industry is fierce. You have some of the biggest names playing ball, cutting IT costs, and providing best-in-class service. The world of IT service management is challenging with ever-changing technology, and it's exciting with innovative ideas coming into play. At the same time, it's a race that can be won only if you couple technology with management.

ITIL Conception

The history of ITIL is nebulous and inconsistent. It started sometime during the late 1980s as a collection of best practices in IT management. A department in the UK government, known as the Office of Government Commerce (OGC), sanctioned the coalition. Basically, the best practices of various IT departments and companies in the United Kingdom were studied and documented. It is believed that most of the initial practices that constituted ITIL came from IBM.

The first version of ITIL was bulky and lacked direction with a compilation of more than 30 books. The second version of ITIL was cut down to nine books in 2000 but mainly revolved around two books: service delivery and service support. The ITIL certifications were based on these two books as well. ITIL v2 introduced ten processes, five each from service delivery and service support. I started my ITIL journey with ITIL v2.

ITIL v2 was process-centric. IT organizations were expected to operate around the ITIL processes. The processes were interconnected but lacked a broader vision and a flow to move things along.

The shortcomings and inadequacies in v2 gave rise to ITIL v3 in 2007. It has 24 processes, spanning the entire lifecycle of a service, from conception up to a point where the service runs on regular improvement cycles.

ITIL v3 came out with five books, each book spanning a lifecycle phase of an IT service. ITIL v3 has penetrated most IT organizations. Even conservative IT organizations have embraced the ITIL v3 service management framework with open arms. The framework is rampant in the industry today and enjoys the monopolistic nature, except for Microsoft, which adheres to a derivative version of ITIL, the Microsoft Operations Framework.

In 2011, ITIL v3 received a minor update where a couple of new processes were added along with some minute changes in definitions and concepts. The latest version of ITIL is referred to as ITIL 2011, and some people refer to it as ITIL v3 2011, indicating the version and the revision year. It's been close to a decade since the new stable version was introduced, and colloquially people refer to it as simply ITIL, without any versions or revision years appended to it. ITIL currently has 26 processes and four functions.

Figure 2-1 depicts ITIL versions over the years.

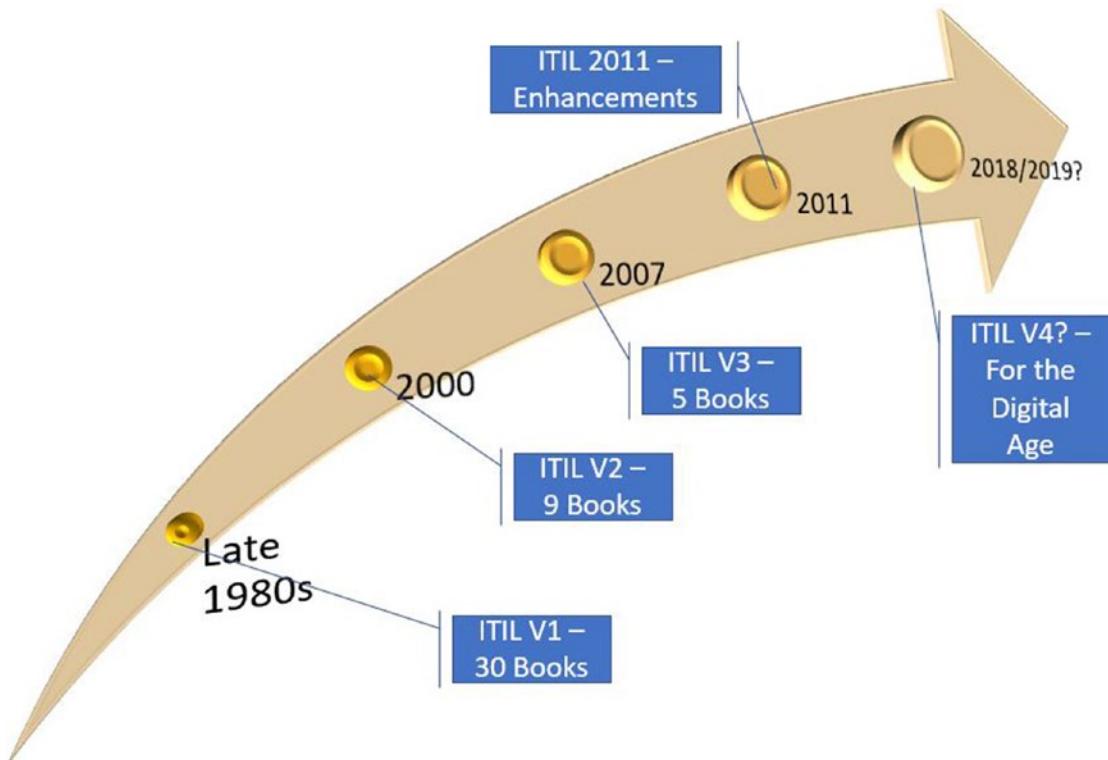


Figure 2-1. ITIL over the years

We are in the age of digital innovation where the changes seem to be happening at a rapid rate. To cater to the digital age, Axelos is working on upgrades to ITIL 2011, which is either expected to come in late 2018 or early 2019.

Competition to ITIL

ITIL has been dominant for the past two decades. There are no other service management frameworks that are competing for space. It is quite lonely in the club of service management frameworks. Why do you think this is the case? A lot of things have worked in ITIL's favor. It has a single objective—to deliver value to the business. To deliver unparalleled value, it has adopted the following characteristics:

- ITIL is based on best practices.
- ITIL is nonprescriptive.
- ITIL is vendor and technology neutral.
- ITIL is nonproprietary.

As mentioned in the previous section, best practices are collated from various organizations. Some organizations may be doing a great job of gathering requirements, while others focus on identifying improvements. So, when you take the best of such organizations and bring that together, you have knowledge that is enviable.

Proprietary knowledge, on the other hand, stays within close quarters, and fewer heads have been banged together to come up with proprietary knowledge, which may be good, but it's not as diverse and experienced as public frameworks such as ITIL.

Proprietary knowledge is developed for the sole purpose of meeting the organization's objectives. It is not meant to be adapted to meet other organizations' objectives. Moreover, if you are adopting proprietary knowledge, you are expected to pay a fee or royalty of some kind. Public frameworks are free. When you can get an all-you-can-eat buffet for free, why would you opt to pay for dinner à la carte?

Service Management in the Digital Age

We are in the digital age now. Everything that we do and process is done in a digital way. We don't have days anymore to make changes; it's now all about minutes and hours. The world is spinning on an axis that turns pretty much the way the wind blows. And what we need are frameworks and methodologies to support the rapid changes and transformations in direction and technology.

DevOps and Agile are being refined to keep the digital age afloat, and the segment that seems to be lagging behind is the service management area, which is the inspiration for this book. ITIL was last refreshed in 2007, and some minor modifications were introduced four years later. Since then, a lot has changed. The area of service management is now seen through the lens of automation, optimization, and nullification. For service management to be effective, it needs to be more adaptive and respond quickly to rapid changes. The good thing about ITIL is that it is not prescriptive. So you can adapt ITIL in principle and set the course for implementation in the digital age. Axelos, the company that owns ITIL, has announced that a practical guidance on using ITIL along with DevOps, Agile, and Lean is coming out in the later part of 2018.

The official practical guidance that ITIL provides for various industry sectors complements the ITIL publications, however, they try not to go too specific into the industry practices. Hence, there is a need for a bridge that connects the ITIL framework with the runway, which is the need that I am trying to fulfill with my book.

Note Verism is a new service management approach that was released in January 2018. It provides guidance on how to adopt ITIL in the digital age. Since it is fresh on the shelves, nobody knows if it is going to be the way forward. We have to wait until it is implemented and then comment on it.

Understanding Services

ITIL v3 is a framework that is centered on IT services. So, it is imperative to first understand the meaning of a service, according to ITIL. Here is the official definition of an IT service:

“A service is a means of delivering value to customers by facilitating outcomes that customers want to achieve, without the ownership of specific costs and risks.”

The best way to understand anything that is complex is to break it into parts. This is my method for understanding the concept of IT services.

The first section of the definition states: “means of delivering value to customers by facilitating outcomes that customers want to achieve.”

IT services in ITIL are defined from a customer viewpoint. Essentially, an IT service must deliver value to the customer. The value delivered must be something that the customer considers as helpful. Let’s take the example of an IT service that is quite common across the board: the Internet. An Internet service delivers value to customers to help them achieve their objectives. So, it fits the bill of what an IT service is all about.

If the Internet service provider (ISP) were to provide speeds upward of 100MB per second for a customer who only checks e-mails, it would be overkill. The high speeds offered by ISPs are generally appreciated by gamers and social networking users. In contrast, the customer who uses the Internet service for checking e-mails does not find any special value between a high-speed Internet and a normal Internet connection. But for a user who hogs a lot of bandwidth, it is valuable. To summarize, the value of an IT service is derived from the customer’s standpoint. So from this example, value to one customer may not be value to another.

Now the last part of the definition states: “without the ownership of specific costs and risks.”

The customer enjoys the service but does not pay for specific costs. Instead, they pay for the service as a lump sum. For example, in the Internet example, the customer pays for the high-speed Internet a fixed sum every month, not a specific price for the elements that make up a service, such as the infrastructure that supports it, the people who maintain and design, and the other governmental regulation costs. Instead, the customer just pays an agreed amount.

The final part of the definition states that the customer does not take ownership of the risks. Yes, but the Internet service provider does. What are some of the risks that exist in the IT world pertaining to ISPs?

- Fiber cuts
- Availability of support technicians
- Infrastructure stability among others

The customer protects himself against the risk of enjoying services through service level agreements (SLAs) to guarantee service at certain minimum levels (fit for use and fit for purpose).

Service Types (Components)

A service can be broken down into three components. Essentially what a service provides is the heart of a service; the component is called as the *core service*. The core service is customer-facing; however, the core service is powered by sets of services called *enabling services*, which is the final component. Finally, the core service is embellished with other service to make it more attractive (the third and final component) called *enhancing services*.

So, these are the components of a service:

1. Core service
2. Enabling service
3. Enhancing service

Figure 2-2 illustrates the three components of a service.

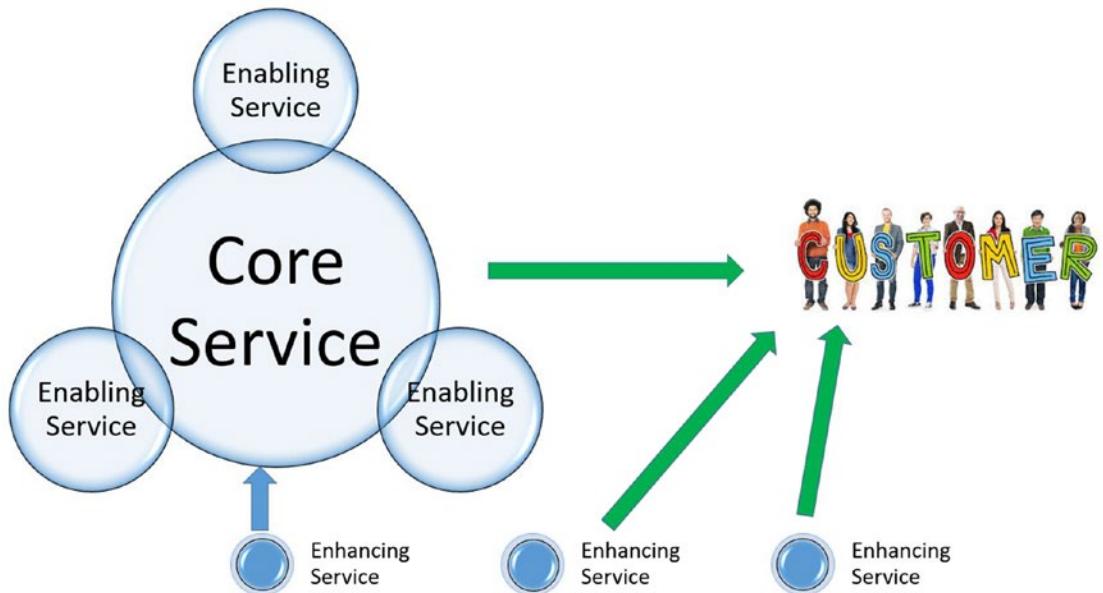


Figure 2-2. Types/components of a service

Core Service

A core service is at the heart of the service that is delivered to the customer. It delivers the basic outcomes that the customer is interested in. The value representation for the customer starts with the core service, and it is the main driver for which the customer is willing to pay for.

For example, say a customer signs on with a cellular service provider to make and receive calls on the go. The most basic service that a cell phone service provider offers is the ability to make and receive calls. This is the core service. If the service provider fails to deliver the telephony functionality to the customer's satisfaction but instead focuses on other add-ons such as high-speed Internet, the customer will still be unhappy because the core service, the reason behind the customer's decision to pay for the service, has backfired.

Enabling Service

For the core service to work, you need services that can support it. These services are the enabling or supporting services. Enabling services are mostly (almost always) not customer-facing, and the customer may or may not see them. The customer does not pay for them individually, but the payment that goes toward a core service gets back charged internally for the enabling service.

For example, sticking with the cellular service example, what services do you think are needed to support the core service of making and receiving calls?

- A service for installing and maintain cell towers
- A network service for routing your calls
- Software service for accounting and billing

This list could get pretty long. I hope you get the idea. All the aforementioned services work in the background and exist to support the core service, without which the cell phone service may not function like it should. So, the customer does not get billed individually against the cell towers, network services, and so on, but rather the service the customer enjoys comes with a price tag attached to it.

Enhancement Service

Enhancing services provide the excitement factor to the customer. They add on to the core service, providing a number of services that most often excite the customer into paying more for the service. The enhancing services may not function on their own, so it is necessary for them to be piggybacked on the core service for their deliverance.

A core service can exist without enhancing service, but the reverse is not possible. The presence of enhancing services differentiates the service provider from others in the market.

For example, the customer can make and receive calls. What else? When looking at the service brochure, the customer was more interested in what else the service provider could offer as the calling part was a given. It offered 4G Internet, Internet hotspots around the city, voicemail, SMS, and others. These additional features helped the customer make a decision in choosing the service provider.

Understanding Processes

ITIL is made up of processes. Just as with services, you cannot get into the nitty-gritty of ITIL if you don't understand the concept of a process. I will give you some examples to emphasize its importance so that your foundation is strong for what you have to build on for your career. For the official definition, ITIL defines a process as follows:

"A structured set of activities designed to accomplish a specific objective. A process takes one or more inputs and turns them into defined outputs."

You can envision a process as a set of activities that you need to perform, one after another, to achieve something. Each activity that you perform sets the precedence for the next one and then the next. The objective of a process would be to achieve an output that is along the expected lines and as desired.

Now for an example to make the concept simple and digestible. A process is similar to a recipe for cooking a dish. In a recipe, you have several steps that you need to follow, as instructed, to get the dish you desire.

Let's look at the recipe for an egg omelet. It goes something like this:

Step 1: Break a couple of eggs into a bowl.

Step 2: Whisk them until they become fluffy.

Step 3: Add salt and pepper to the mixture.

Step 4: Heat a nonstick frying pan, and melt some butter until it foams.

Step 5: Pour the egg mixture into the pan, and tilt the pan until it covers the base.

Step 6: Cook for a minute or two, and flip the omelet and cook it for a minute more.

Step 7: Serve the omelet hot with toasted bread.

You need to follow the steps to get an egg omelet. You cannot interchange any two steps to get the same output. In IT language, this is the *process* to make an egg omelet.

The main aspect of a process is the interconnectivity between the individual steps, and collectively, all the steps work toward a common goal, or a common objective that is desired.

Note All processes must mandatorily have an input, a trigger, and an output. A trigger will initiate a process to kickstart, and the provided inputs are processed to provide a predictable output. In the example provided, eggs are the input, and hunger or the need to have breakfast is the trigger. The finished product of the omelet is the output.

Understanding Functions

Before I discuss functions, let's take a look at organization structures. It is quite common these days for there to be teams with people who have expertise in one area. Examples could be the networking team, the Unix team, the Windows team, the Java team, and the web development team. It is also in vogue that teams are carved out based on the depth of knowledge. An example would be a Network L1 team (junior), Network L2 team (senior), and Network L3 team (expert teams, the architects). L1 teams consist of people with less experience, and the tasks they are asked to take care of are quite basic and administrative in nature. For an L2 team, it gets a little more complicated, and they could be asked to troubleshoot and diagnose outages. An L3 team could be your top-notch team that not only provides support when L2 needs it but also helps architecting networks.

The teams that I have been referring to are known as *functions* in ITIL, nothing more, nothing less. There are only four functions that are defined in ITIL, and all of them come into play during the entire lifecycle of ITIL framework. The official definition of a function is as follows:

"A team or group of people and the tools or other resources they use to carry out one or more processes or activities."

Functions in ITIL

All the functions are defined in the service operations publication. The list of functions is as follows:

- Service desk
- Technical management
- Application management

- IT operations management
 - IT operations control
 - Facility management

Processes vs. Functions

There are processes, and there are functions in ITIL. While there are 26 processes, there are only four functions. Processes don't run by themselves. They need people to carry out the individual process activities in traditional ITIL (you will find out later how processes can be automated later in this book in DevOps). And the people the processes look for come from functions. To state it simply, functions provide the resources needed by the processes to complete their objectives.

Within the organization where you work, there are verticals—say banking, retail, and insurance. There are processes that cut across all the verticals of the organization such as human resources. The people in the verticals perform their role in the human resources process, which is horizontal cutting across all verticals, even though they are part of a function. This would be an example of how a process leverages functions for carrying out the set objectives.

Figure 2-3 illustrates the intersect between processes and functions. You can replace the functions with the verticals in your organization and the processes with the common processes such as travel process, promotion processes, and others to establish a better understanding.

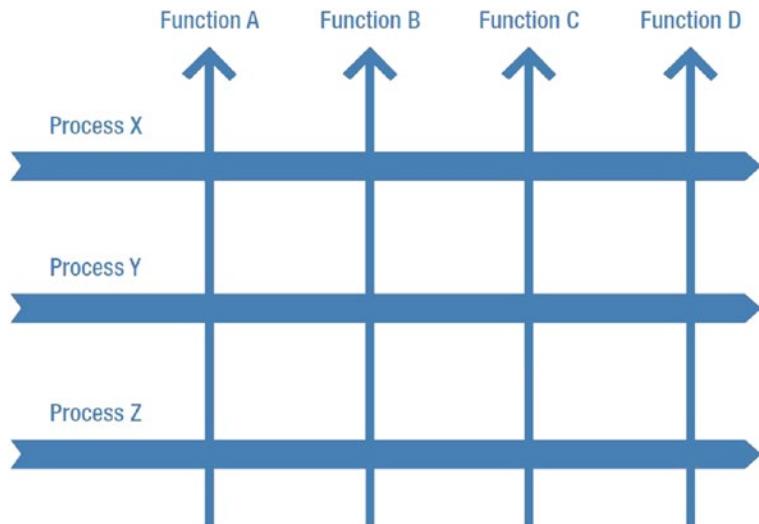


Figure 2-3. Intersect between processes and functions

ITIL Service Lifecycle

ITIL v3 was derived from various high-level activities that encounter an IT service, and each of these high-level activities was introduced as phases in the ITIL service lifecycle. The five phases are as follows:

1. Service strategy
2. Service design
3. Service transition
4. Service operations
5. Continual service improvement

These five phases are represented in Figure 2-4. It shows service strategy at the core to indicate the importance and involvement of a sound strategy in the inception of IT services. Service strategy provides guidance around existing and new IT services. Surrounding service strategy includes service design, service transition, and service operations. Service design provides the direction pertaining to the realization of a service. The IT services that are identified in the service strategy are defined and designed, and blueprints are created for its development. These designs are built, tested, and implemented in the service transition phase. After implementation, the services move into a maintenance mode. Maintenance of services is handled by the

service operations phase. Continual service operations envelop the other four phases. The depiction shows that all four phases present opportunities for improvement, and the continuous service improvement will provide the means to identify and implement improvements across the service lifecycle.

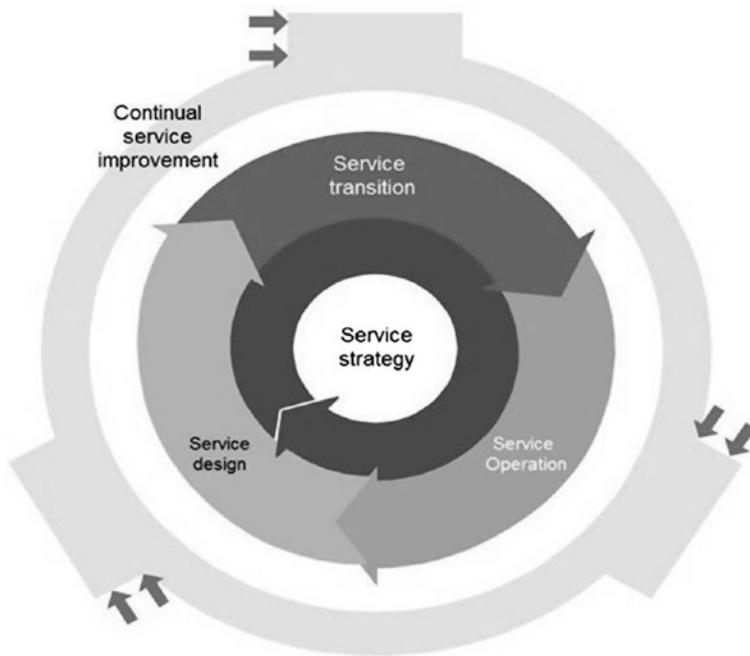


Figure 2-4. ITIL service lifecycle

Did you notice that every single phase in ITIL has *service* in it? This is not happenstance but rather strongly indicates that ITIL is service-centric, and it revolves around services that provide value to customers.

Service Strategy

Service strategy is at the core of IT services. It is the heart of service management. The main intent of ITIL and IT service management is to create value for customers. The value creation starts at the service strategy lifecycle phase.

In this phase, the question “Why do it?” is posed before the question “How to do it?” The core intent of this phase is to develop a strategy and create services that add value to customers. For the service provider organization to flourish financially, everything must be business as usual at the end of the day.

Note Business as usual is commonly abbreviated as BAU. It refers to employees of an organization completing their respective day-to-day tasks in a normal way. Exceptions and ad hoc changes to the work done does not count as BAU.

Let's say that the major occupation of most residents in a city is IT, and their staple food is noodles. As most people in IT spend several hours at office, they find it difficult to fix food at home. So, they end up ordering takeout or cooking with instant noodles. Off late, there has been bad press against instant noodles, and eating out every day is not working out very well for the residents financially; in addition, health concerns are beginning to dawn. What will be a good product to introduce in this market?

A gadget that allows the residents make their own noodles would be a godsend, where the residents have to add flour, oil, water, and other ingredients into the machine. And at a click of a button, if the gadget can start churning out noodles (healthy because the raw materials were produced by residents themselves), then the chances of the company that manufactures this gadget becoming an instant wonder is quite bright. All that the company did was strategize to find out what their market lacked and filled the void with a solution. Instead, if they had marketed this product to a sleepy town where men and women have ample time on their hands to fix their own food and healthy noninstant noodles are available on every shelf across the stores in the town, then the same company would have had to shut shop quite early in the game.

In short, strategy is not a silver bullet that a best solution will be an elixir for all problems. Every problem will have a different solution, and identifying this solution is a strategy that is bound to make or break companies.

The most important aspect of service strategy is to understand the customer, identify customers' needs, and fill those voids. If the provider can do this, even a dim-witted service or product would take off exponentially, until someone else finds a competing service or a product to counter yours.

To explain with another non-IT example, let's say that a landowner identifies a location in a popular neighborhood that lacks a decent mall in the vicinity. Building one would be like striking gold; you would have customers waiting to lap up what you have to offer once it is built. This move can potentially be termed as a successful strategy.

Specifically, with ITIL, service strategy's role is to provide guidance on creating value through IT services. The idea is to introduce services that have the potential to succeed and garner market.

Service Strategy Processes

The following processes are listed in the ITIL service strategy publication:

- Strategy management for IT services
- Service portfolio management
- Demand management
- Financial management
- Business relationship management

Service Design

At the end of the service strategy lifecycle phase, smart heads (read: leadership) have thought over and have provided direction and guidance on which services to offer. The outcome of the service strategy is like the idea that an entrepreneur comes up with. Whether the idea will come to fruition will become known in time.

The service design lifecycle phase answers the question “How do I do it?” It takes the idea and comes up with solutions and designs that give wings to the ideation process set forth in the previous phase.

The success of a service depends primarily on the service design phase. While strategy plays an initial part, the solution to make it happen is equally important.

Tablet computers have existed for a long time. My earliest memory of one was the Palm Pilot in the 1990s, and I started using Windows-based tablet PCs in the early 2000s. They were commonly called *personal digital assistants* (PDAs). But it was not until the introduction of iPads in 2010 that led to an explosion in the demand for the touch-capable portable computing devices. They stepped in and became synonymous with tablet computers.

What are the differences between an iPad and all the other personal handheld devices that came prior? In my opinion, it is the iPad’s design that made the difference. The strategy was out there since the 1980s, but the lack of a good design didn’t take the early versions to the pinnacle as you would expect. This is my interpretation of the tablet computer history and its relation to design; others may see it differently. The key takeaway is to highlight the importance of design. Before I end this topic, I will honorably mention Android tablets, which are competing with the iPads neck to neck. There is nothing better than two good designs fighting for fame on a strong foundation built on wise strategy.

Revisiting the non-IT example that I used with service strategy, after identifying the location, the landowner will have to hire the best architects to bring the most value money can buy on the land that is most sought after. The architectural designs that the architects come up with become the blueprint that gives insight on paper as to what things will look and feel like once they are realized.

Service Design Processes

The following processes are listed in the ITIL service design publication:

- Design coordination
- Service level management
- Availability management
- Capacity management
- Supplier management
- Information security management
- Service catalog management
- IT service continuity management

Service Transition

The output of the service design is a set of design documents giving you the designs pertaining to all aspects of a service. The next task is to develop the service based on the designs. In the ITIL world, this is called the *service transition*, where the designs are transitioned into production environments through development, testing, and implementation.

The service transition lifecycle phase answers the question “What do I develop and deploy into production?” To achieve the objectives of service transition, you could employ either service design lifecycle activities, hardware delivery lifecycle (HDLC) activities, or any other framework that delves into building a system/service and deploying it into the intended environment. ITIL is flexible like that; it can integrate seamlessly with any of the frameworks you can throw at it.

Going back to the non-IT example, there are architectural drawings from the previous design phase. These designs are handed over to a qualified builder to construct the mall as per the architectural designs. The builder constructs the mall in this phase as per the plan and brings it to a state where it could be operationalized. This is exactly the role of the service transition phase.

Service Transition Processes

The following processes are listed in the ITIL service transition publication:

- Change management
- Release and deployment management
- Knowledge management
- Transition planning and support
- Change evaluation
- Service validation and testing
- Service asset and configuration management

Service Operations

Service operations is the most popular phase of the ITIL service lifecycle. The reasons are twofold.

- Operations run for a long time. I am trying to avoid the word *infinite* here, as there is nothing guaranteed in this world. So, in effect, operations run for a long time, which translates into most service management practitioners working on the service operations lifecycle phase.
- As the phase runs the maximum amount of time, it has the maximum number of touch points with the customer. Moreover, operations is considered to be the first point of interaction for a customer on a regular basis.

Service operations entails maintenance and making sure the services are running as per the plan—the status quo is achieved. Under service operations, there are no new development or deployments, only maintenance. When I say no deployments, I will clearly differentiate that from regular patching or some releases being deployed for maintenance issues. Some maintenance activities could include doing health checks, fixing issues when they arise, and ensuring recurring activities are scheduled and run as planned.

Drawing on the previous example, the mall owner takes possession of the mall, rents out the shops, and sets the ball in motion for it to run smoothly. For it to be operationalized, he needs to hire people who can manage various areas of the mall and employees who can carry out day-to-day tasks, such as cleaning, security, marketing, and so on. He also needs to set up daily/weekly/monthly/yearly activities as required activities to keep the mall functioning. Examples could include monthly generator checks, security audits, four-hour restroom janitorial services, etc. Do you get the drift?

By looking at this simple example, you can easily see the activities that are needed in operations. The operations phase in IT service management is a lot more complicated and requires plenty of minds to work out various aspects of it.

Service Operations Processes

The following processes are listed in the ITIL service operations publication:

- Incident management
- Problem management
- Event management
- Request fulfillment management
- Access management

Continual Service Improvement

The final phase in ITIL is continual service improvement. While I call it the final phase, it does not necessarily come into play after the service operations phase. If you look at Figure 2-4 closely, you will observe that this phase encircles all of the other four phases. There is meaning to this. This phase takes input from any of the other phases to carry out its process activities. You can also say that it does not fit in the lifecycle phases

because it does not roll once the previous phase has completed its delivery, but will feed improvement opportunities to the previous phases. But remember that this is the phase that keeps the ball rolling, or the service breathing.

I strongly believe that if something does not grow, it is as good as dead. This is true with our careers, bank accounts, or anything else you might think of, except of course our waists! This concept applies to services too; if they do not improve over time, IT services wither away and something else takes their place. The objective of the continual service improvement (CSI) phase is to identify and implement improvements across the four lifecycle phases; be it improvements in strategies, designs, transition, or operations, CSI is there to help. It is also the smallest phase of all the phases in ITIL.

In keeping with the example, in the fully functional mall, you might have thought that general maintenance should be sufficient for upkeep and ongoing operations. This may be good for a brief period but not for long. Other malls are competing with it in terms of amenities, parking availability, and aesthetics, among others. If our mall does not improve in due time, customers are going to lose interest, and sales will start to dwindle. So, to keep up with the growing demands, the mall owner must find ways to make the mall exciting for shopkeepers as well as for customers, perhaps providing space underground for a public transit station, valet parking for certain customers, free high-speed Internet for customers, and installation of moving walkways.

These improvements need not happen overnight; it can be a process that takes place over days and months. But the important thing is to keep improving the mall on a regular basis.

Continual Service Improvement Process

The following process is listed in the ITIL continual service improvement publication:

- Seven-step improvement process

Note It takes an extremely mature service organization to implement all five phases. Generally speaking, service design, transition, and operations are the most often implemented phases, followed closely by continual service improvement. Service strategy is sparse.

ITIL Roles

ITIL is a harbinger of employment. It has introduced a number of roles, all useful and necessary, that are the most sought after in the IT industry today. As mentioned earlier, ITIL has 26 processes, and each of these processes needs to be owned, managed, and practiced. Automation has its place in ITIL, but machines cannot do what people can, even in the age of machines ruled by Skynet!

Note According to Wikipedia, Skynet is a fictional neural net-based conscious group mind and artificial general intelligence system that features centrally in the *Terminator* movie franchise and serves as the franchise's main and true antagonist.

Every ITIL process brings to the table at least a couple of roles (process owner and process manager). So, it brings plenty of employment opportunities, plus customers would be happier dealing with people with the right skill set and with the organization that has clarity over people owning and managing respective areas. So, with 26 processes in the pipeline, you are looking at more than 50 distinct roles at a minimum.

At a framework level, there are four roles that can be applied to various services and processes. The roles are that of a service owner, process owner, process manager, and process practitioner.

Service Owner

Earlier in the chapter, I explained what a service is. So, this service, which provides value to the customer, must have an owner to ensure somebody has accountability. The person who owns the service from end to end and the person without whose consent no changes would be done is the service owner.

In the mall example, the mall owner is accountable for the shopkeepers and the customers. He owns the place, so he puts his signature on all changes being made to it; in other words, he approves enhancements and modifications and decommissions if any. He is the service owner in ITIL terminology.

Process Owner

A process is a set of coordinated activities that exist to meet the defined objectives. This process, or the series of coordinated activities, needs an owner, someone who has a finger on the pulse to check whether the process is fit for the purpose and that it is subjected to continuous improvements.

This person is the process owner and is accountable for the process deliveries, be it in terms of effectiveness or efficiency.

In the mall example, there will be several processes defined and implemented. One such process is the process to maintain diesel generators. The maintenance process could go something like this: weekly general checks on Sundays at 10 p.m. and detailed monthly checks on the first Sunday of each month at 11 p.m. Checks are done based on a checklist. If minor repairs are identified, they are carried out during the maintenance window. If a major repair is identified, a suitable window is arranged, all the necessary resources are mobilized, and repairs are carried out by a specialist team. This diesel generator process cannot be orphaned. It needs somebody to own it and ensure that it is meeting its objective, which is no outages because of the generators.

Process Manager

You know what a process is and who the owner is. It is unlikely that an owner will actually manage things on his own. He will hire people who can manage the process for him.

Process managers ensure that the processes run as per their design and achieve what they're meant to. Since they are close to the work, they are in a good position to suggest improvements to the process owner. A decision to accept or reject the suggestions is made by the process owner.

A process manager is accountable for the operational management of the process, which means coordinating activities between various parties; monitoring, developing, and publishing reports; and, as mentioned earlier, identifying improvement opportunities.

In the diesel generator maintenance process, the process owner hires an electrical engineer to manage the maintenance activities and to report on the outcomes. The maintenance manager is responsible for ensuring that the technicians involved have the right skill set and are following the right set of instructions in carrying out the maintenance activities. If the manager finds that the weekly checks are not adding value,

he can suggest to the process owner to shelve the weekly checks and schedule them for every two weeks. As mentioned earlier, the decision to make the checks every two weeks is made by the owner, not the manager.

Process Practitioner

Anyone who plays a part in the process is a process practitioner. This may be the manager or the owner or someone who may not be part of the process hierarchy. To rephrase, people who are responsible for carrying out one or more activities in a particular process are process practitioners.

In the generator maintenance process, technicians have the responsibility to check the generators based on a checklist. They are process practitioners. It is also likely that the technician is a process practitioner for multiple processes, depending on the number of processes he is acting on. For example, he could also be responsible for electrical maintenance, electrical repairs, and elevator maintenance, thus being a process practitioner in each of these processes.

RACI Matrix

In an organization, it is important that roles and responsibilities be clearly defined. When there is ambiguity over responsibilities for activities, it often leads to inefficiency within the system. You might have seen in your own organization that a lack of clarity over roles and responsibilities can end up in a mess, where both of the perceived responsible parties duplicate activities or both leave them for the other to act on.

RACI is an acronym for Responsible, Accountable, Consulted, and Informed. According to the ITIL service management framework, these four types of roles can be used to define all responsibilities and ownership in an organization.

Responsible: The person who is responsible for carrying out the activity gets this tag. He is the person who actually completes the work. Examples could be your process manager and process practitioner, who are responsible for managing activities and performing deliveries, respectively.

Accountable: This is the person who owns the activity. He is the person who is the decision-maker. Examples are the service and process owners. It is important to remember that although in the

real world you could have joint ownership, in the world of ITIL, there is no joint ownership. An activity has a single owner. It can never be shared by two individuals.

Consulted: In any organization, you have subject-matter experts who need to be consulted before and during activities. These people play the role of a catalyst in the service management organization. They do not own anything, nor do they get their hands dirty in the actual operations. But, they provide their expertise in the successful execution of the activity. Examples are corporate lawyers and technical architects.

Informed: There are the people who just like to soak in the information. They do not have any role in the activity but would like to be informed of the progress or the lack of it. They are, in other words, stakeholders without the power of making decisions. Examples are users and senior management.

An Example to Understand RACI

Table 2-1 shows an example of how a RACI matrix looks. It has activities to be performed as part of a process in several rows. Those who play a role in the process make up the columns. You get a matrix by putting the activities and the roles together.

Table 2-1. RACI Matrix Example

Activities	Mall Owner	Maintenance Manager	Maintenance Engineer	Customer
Schedule maintenance activities	C	AR	I	
Sponsor maintenance activities	AR			
Perform maintenance activities		A	R	I
Communicate to customers	A	R		
Fix issues with diesel generator	I	AC	R	

In the example, the activity “Schedule maintenance activities” is owned and performed by the maintenance manager (AR represents Accountability and Responsibility in the respective cell). So, both the accountability and responsibility lie with him. For this activity, he is consulting (represented by C) with the mall owner on suitable dates and informing (represented by I) the maintenance engineer on the maintenance schedule.

Let’s look at the final activity: “Fix issues with diesel generator.” In this activity, the accountability lies with the maintenance manager, but the person performing the fixing is the maintenance engineer. The engineer consults with the manager regarding this activity, as the manager is experienced in diesel generators. The mall owner is merely informed of this activity.

Tips on RACI Creation

Developing a good RACI matrix takes experience and good insight into the activities on hand. However, there are a few ground rules that will aid you in your RACI creation endeavors.

- For every activity, you can have only one person accountable.
- Responsible, consulted, and informed can be spread across multiple roles, although I have not illustrated this in the example.
- A single role can don various hats, such as accountable and responsible for “Sponsor maintenance activities” by the mall owner.
- Accountable and responsible are mandatory for every single activity.
- Consulted and informed are optional. If you are not informing anyone of an activity, you may not have the informed role for the particular activity. “Sponsor maintenance activities” is an example.
- Identify and document as many activities as possible in the RACI matrix, as long as the activities have specific deliverables coming from it.

How Far Is ITIL from DevOps?

This is a million-dollar question that no ITIL expert will be able to answer right away. Every organization implements ITIL in its own sweet way. Likewise, DevOps implementations can be done in a number of ways and with a scope that can be as narrow as introducing automation to a certain task or as wide as implementing DevOps completely. So, the distance between the two is rather unimportant from a theory perspective.

However, what is important is to understand the commonalities and conflicts between ITIL and DevOps. Identifying this gives us the ammo to use the commonalities to our advantage and to foresee conflicts before they happen.

The next chapter delves into the differences between ITIL and DevOps, and the rest of the book is dedicated to unraveling the beauty that can emerge from the synergy created through the union between the two.

CHAPTER 3

ITIL and DevOps: An Analysis

In Chapters 1 and 2, DevOps and ITIL were introduced as an independent framework/methodology, and I consciously did not make an effort to bring together the two entities. I will analyze them together in this chapter. I will dissect the big-ticket conflicts, and the chapter will set the tone for the rest of the book to follow, in other words, for creating a union between the two rather than replacing one with the other. Chapters 1 and 2 clearly specify that the two are leaders in their own respective areas, and one cannot reasonably replace the other. The question of replacement has arisen because the newer of the two, DevOps, steps on ITIL's foot and claims to take over certain aspects of service management.

Today the reality is that ITIL is (almost) everywhere in some form or the other. The majority of development projects are running on Agile today, and the normal extension of it, DevOps, is being talked about in great length; however, the implementations are far and few. One of the prime reasons is because of the ambiguity that exists between product development and service management. Wherever DevOps is being implemented (with the exception of the Amazons, Nexfixes, and others I discussed in Chapter 1), it is being restricted to the development side of things alone. The time is right to make DevOps whole by removing the uncertainty and ambiguity (see Figure 1-4 in Chapter 1), and that is the reason for this book's existence. The book is named *Reinventing ITIL in the Age of DevOps*, and it's a practical approach to implementing ITIL in DevOps projects. However, this book is so much more than what the title suggests because it realizes the DevOps principles to the fullest and will make DevOps a potent force for years to come.

Product vs. Services

Traditionally speaking, the IT industry is like a magnet that has two opposing poles: products and services. So, you are running either a development project or a maintenance one. The people involved on the development side of things have different skillsets, and the support personnel has their own unique talents.

Figure 3-1 indicates this IT magnet, featuring products on the left and services on the right.

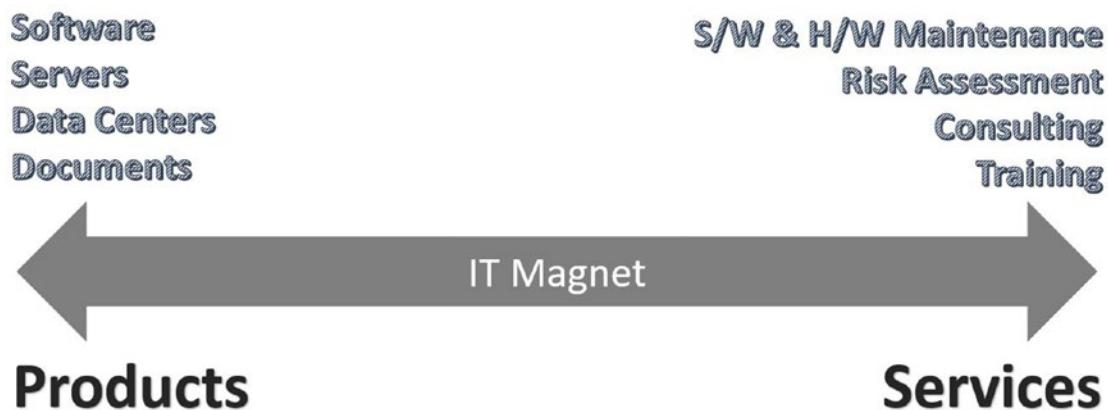


Figure 3-1. *IT magnet*

Products typically are those that are tangible—in a digital way, meaning there is a digital/physical deliverable that is produced. They can be consumed either now or at a later point in time. Services, on the other hand, are intangible—it is the experience that is the result of consuming a service. More important, a service is consumed in the present, rather than getting serviced for future needs. I am not referring to purchasing services for an advanced period of time but rather experiencing something. For example, you can head to a bakery and purchase a quiche. That's a product. You can choose to eat it either now or later. The experience of eating it can be experienced only when you bite into it. This is service.

Referring to Figure 3-1, some examples of products include software, servers, data centers, and documents. All the products can be purchased one time and be used now or later. On the other end of the IT magnet is the services. Maintaining the software and the hardware that is either purchased or developed falls under services. Other examples include conducting risk assessments, consulting, and training. All the services mentioned can be experienced only in the present.

The product industry depends heavily on the service industry because the products that are developed or purchased need servicing on a regular basis. Take, for example, a car that you purchased this summer. You paid the dealer a certain amount of money and purchased the car. The transaction between a consumer and a product manufacturer for all practical purposes is done. This car, however, needs to be maintained on a regular basis. For somebody not so hands-on like me, I depend even more on the auto service stations to maintain the oil and other fluid levels in my car and to check the various parts of the car every six months. The auto service station, as the name indicates, is a service provider that gets business every time I choose to experience the service. There is an open debate whether the product industry is more profitable or whether it is the service industry given that the service industry comes into play as long as the product is relevant. The other school of thought is that the product replication and the profit by numbers make the product industry a better proposition. I don't see this debate getting settled anytime soon!

The reality is that although we have products and services, the gulf between the two is fast shrinking. The ideological differences between the two industries are not going well with the consumer who is facing the heat from the divide. The cost of producing products and living off the profits is no longer feasible as there are intense price wars between competitors. For the service provider too, the costs of providing services are rising, but the market conditions do not allow them to charge higher rates to the consumers. Neither the product manufacturers nor the service providers are able to survive in the present market. So, how do they survive?

The answer is that the product and service industries bring together their resources and capabilities to become a solution provider, as illustrated in Figure 3-2.

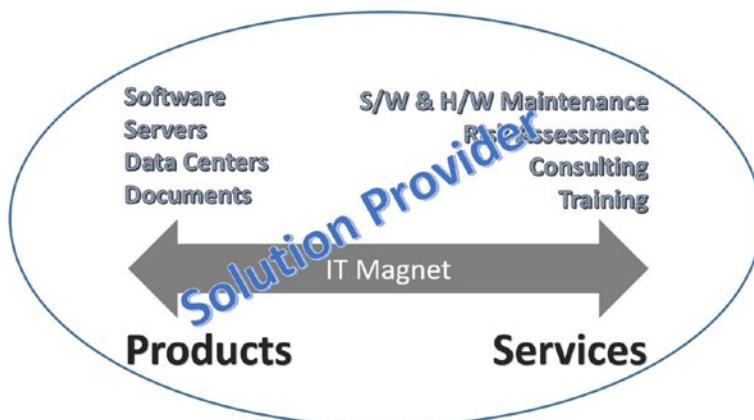


Figure 3-2. Solution provider

How does it work? Does it mean that an organization that develops products starts providing support as well? Not really. This model has been in place for some time where the product manufacturer extends support for their products for a certain period of time. This is a failed model; in addition, this organization is nothing but two entities (products and services), and the gulf instead of existing between different organizations exists within the same one.

A solution provider brings together products and services with an aim to solve the needs of customers. The solution is developed based on the problems faced by the customer and the specific requirements. In other words, it is customized to ensure that the customer gets focused value from the overall delivery of products and services and at an optimized cost.

Today you will find that the gap between products and services is quite narrow, and the solution to customers' needs and problems is being addressed by conjoining products and services. The outcome of this marriage is popularly known as XaaS—which refers to anything as a service. Specific popular examples include software as a service (SaaS), platform as a service (PaaS), and infrastructure as a service (IaaS). In the XaaS model, every need of a customer is turned into a service, including products. So in essence, the marriage of products and services has given birth to the *product as a service*.

There are several advantages of the XaaS model. Referencing Chapter 2 on the topic of services, the definition of a service is as follows:

A means of delivering value to customers by facilitating outcomes that customers want to achieve, without the ownership of specific costs and risks.

In the XaaS model, the customer gets to enjoy the product without the ownership costs, the need for constant upgrades, and the risks associated with it. In exchange, the customer will pay regular rent to use the product as a service. Office 365 is an excellent example; it falls under the SaaS model. I get to use all the programs that fall under Office 365 for a regular monthly or yearly fee. When Microsoft upgrades it, I get the upgrades for free. To start using the Office product, I don't need to spend capital up front, which is the traditional model before Office went the SaaS way. I used to purchase Office products, and within a couple of years, my version was expunct, and I was forced to purchase a new license or upgrade the existing one at a cost.

Startup companies in need of servers and datacenters find the IaaS option most attractive because they can reroute their precious capital into other parts of the organization rather than spending it all on infrastructure. Instead, they pay a monthly or a yearly fee.

Turning our attention back to DevOps and ITIL and their mutual existence, the solution is to bring the two together to get the best results. DevOps can be seen as the products side of the IT magnet, and ITIL is definitely on the services side. DevOps is the right approach in providing the solution to customers rather than products, but for it to happen, there is a need for a solid service management framework such as ITIL. The XaaS model works if the product management and delivery are powered by DevOps and the ongoing services through ITIL.

We don't have other options. The best in the industry models must come together to boost the IT industry from sinking in wastage and to provide the customer with the best possible outcome. The value to customers is based on their perceptions. ITIL identified this aspect a long time back and has put in place various service management parameters that manage the customer expectations, thus ensuring customer satisfaction. The volatile market conditions and the blitzkrieg of technological advancements require a software development process that can turn on its head as the wind changes direction. This is made possible through DevOps implementations. Finally, the XaaS model has greatly benefited the solution providers as well as customers—it's a win-win situation. From all fronts, there is a desire for *Captain Planets* to emerge from the combination of various natural elements, and this is exactly what I am trying to accomplish in this book by bringing the two magnetic poles together and creating a bond that's bound to take the IT industry to the next decade.

Big-Ticket Conflicts

Not all is well with how things stand at the moment between DevOps and ITIL. Importantly, the conflicts between the two must be identified before going into solution mode (Chapter 4 onward). ITIL has been around for a while now, and DevOps is the new kid on the block. So, just we have generation gaps among people, it exists in frameworks as well, leading to certain big-ticket conflicts that I am going to cover in this section. The minor issues are a culmination of the major conflicts; therefore, I expect them to be resolved through dialogue and seeing the bigger picture.

Which Is It: Sequential vs. Concurrent?

ITIL was conceived in the age of waterfall management, and DevOps is anti-waterfall—meaning Agile in nature. The five phases of ITIL are strictly sequential in nature. Unless you have a service strategy, you will not be able to develop new services. Unless you have defined services, you cannot think of designing, building, and implementing them. Only after they are implemented and handed over formally can they be maintained. So, the fruits of the labor are visible only during the service operations phase because the rest of the time, the service is in its development stages and adding no value to the customer.

This is the point of inflection as far as DevOps is concerned. DevOps is based on Agile, which is antisequential. Therefore, DevOps expects some form of output right away. How can we do it in ITIL given that the service developmental works are a long consummated process? As I mentioned earlier, this is the time for identifying conflicts and not resolving them.

Let's Discuss Batch Sizes

With the sequential nature of ITIL, the outcome is one big piece of delivery that contains all the pieces of the puzzle that are required to solve the IT service conundrum. This worked fine for services for the most part, but product delivery today is mostly dealt with in small batches. Every sprint delivers a piece of software that can be independently tested, demonstrated, and verified. The suspense around what is going to come out at the end of the long-winded cycle does not exist in a DevOps-run project.

The longer the batch size, the riskier the proposition is. What if the requirements were misunderstood all along and in the end you produce something the customer never wanted? Either it would be too late to go back and make changes or the road to completion would have plenty of rework. Either way, the customer ends up being not so satisfied. This is where DevOps pitches in with its small batches. If there are any changes to be done, you make them before you start.

It's All About the Feedback

Delivering small batches alone to find out whether the direction you are taking is right is not enough. You need the feedback to come through rapidly so that you are on the right course.

In ITIL, there is never any talk of formal feedback until you get to the final phase, which is continual service improvement. I am not suggesting that people who follow ITIL don't take feedback seriously, but I am pointing out that the framework has missed an important element that's one of the core pillars for satisfying customers.

The small batches that are produced in a DevOps cycle are followed by a feedback cycle where feedback is sought every step of the way, and it comes in the form of test results, acceptance tests, and demonstrations to the customer. Considering that the two-week sprint cycle is most common, even if you were going off-course, the amount of rework that you may be expected to perform would be equivalent to the two-week sprint. This too is arrested by the presence of a product owner who is from the customer's organization and who is part of the team and knows exactly what the team is working on at any point in time. By contrast, in ITIL where feedback is not formalized until you hit the improvement cycle, if the service development takes about a quarter to half a year, the entire efforts will be wasted if it's not as the customer demanded.

The Silo Culture

The ITIL service management framework defines functions that are teams where people with similar skill sets are housed. In other words, every team represents a silo with people with similar talents. As and when they are required to work on process activities, they get pulled into the process role, and then they are sequestered back to their homes. There is the logic behind this. People with similar skillsets, when housed together, can help each other in terms of capability improvement and can work as a team to deliver.

In DevOps, as covered in Chapter 1, DevOps teams are strictly cross-functional and are formed around a product or a project. They are aligned to the product or project alone, and this helps in ensuring that all the right people needed to build and support a product are sitting in the same group and not spread across the organization with differing lines of reporting structure. By housing the cross-functional teams together, invariably teams will understand all aspects of the product, and this will help in dealing with issues and other parts of delivery.

Clearly, ITIL and DevOps are divided over how the team structure should look. It definitely makes sense to put all the similar talented people together to ensure skill development and capability improvement. But is this structure focused on the product and the customer? Is everybody on the team intimately close to the product to ensure speedy delivery? These are some loopholes that are clearly present in the ITIL world.

It makes sense to build cross-functional teams for getting to know the product better. If they are part of the team since inception, the intimacy with the product deepens and helps in supporting the customer in an effective manner. However, does everybody on the team have a role to play throughout the lifecycle of a product? That is the answer to seek! The answer will be along the lines of *no*. So, what do you do with them? In IT, we don't bench people as we did a decade ago. How do we ensure that the intimacy with the product is maintained and yet the resources are utilized optimally? This is a challenge that comes up often in DevOps projects, and the answer lies in a meta-team.

What Is Configuration Management?

The word *football* does not mean the same sport around the world. In the United Kingdom and many other countries, it refers to soccer that is played with a perfectly round ball. In the United States, however, football is a different game altogether played with an oval-shaped ball and is somewhat similar to rugby. Likewise, the word *configuration management* has different connotations in ITIL and DevOps.

In ITIL a *configuration management system* (CMS) is a collection of databases of all the service-related data. It contains configuration management databases, incident records, problem records, change records, known error databases, and everything else that goes into the service management system.

The word *configuration management* is commonly used to refer to source code management (SCM) in the software development industry, and it has trickled down to the DevOps scheme of things as well. The repository consisting of source code, the strategy, and implementation around branches and the way updates are done to the repository come under the purview of configuration management in DevOps.

So, it is quite apparent that configuration management has different meanings in both areas. So, when you try to bring the two systems together, which one would you probably go with? The ITIL configuration management has critical configuration management database (CMDB) information, which is considered as the foundation for setting up services. DevOps configuration management refers to the source code itself, which is at the heart of software development. So, it is not like we can replace one with the other. This is a serious conflict that requires an amicable solution if we are to plan a long game with the two frameworks.

Continuous Deployment Makes Release Management Irrelevant

As introduced in Chapter 1, the continuous deployment process directly deploys the package into the production system when all tests are green. This is the domain of control for the change and release management processes, and by deploying directly without any governance and controls, there is a perception that the change and release management processes tend to become irrelevant.

The ITIL change and release management processes are a trusted set of processes that ensure that the changes going into production are well-tested, and it also brings together all the stakeholders in a huddle to decide the merits and demerits of a potential change. Careful planning around changes by all stakeholders can potentially ensure that the changes are beneficial to the organization and do not disrupt any of the existing setup unless they are designed that way.

By going around these governance processes, DevOps through the continuous deployment process is setting a bad precedent that is uncontrolled and perhaps malevolent to the ecosystem. The general perception is that the development teams rejoice over fewer governances, and the operations teams feel antsy about this whole DevOps thing! The market we are in today too roots for speed and dynamism. In fact, more and more organizations are withering away when they are unable to cope with the speed of change. So, organizations have opted to ride the DevOps rollercoaster and take chances with rapid deployments. In fact, continuous deployment may not be feasible for all types of organizations. Some, such as banks and other financial institutions, have legal and regulatory approvals in their workflow that can hold them back by opting for trigger deployments rather than continuous deployment.

Continuous deployment going around the approvals of change management is a premature judgment call. ITIL change and release management provide the precious governance layer around the changes going into the ecosystem. It would take an artist to bring both together and stitch them up seamlessly, and I have taken up the cudgels to be that artist. There are ways to rein in continuous deployments through the change and release management processes, and we are going to look at them in detail in Chapters 9 and 10. The truth is that we need both. Continuous deployment is the future, and it differentiates market leaders from the rest.

Union of Mind-Sets

DevOps is a philosophy. It's a new culture. With it comes a new mind-set. The DevOps mind-set aims to quicken software delivery through collaboration and working in small batches, aided by automation. The thought process is that working swiftly in smaller batches will help deliver in increments, and even if something were to go wrong, the damage done would be trivial. Rolling it back will be a simpler task as well, with a good chance that nobody on the other end (read: end users) even noticed.

Although never mentioned thus far, service management (aka ITIL) has a mind-set too. The framework is built heavily around the customer, and the mind-set is around creating value to the customer through services. ITIL ensures that a customer's perception of value takes precedence over what it really is, and customers always get what they want. Across the five phases, ITIL ensures that value creation is at the heart of services, and this is a great sign of things to come when we place the service management framework over the DevOps mind-set. DevOps too is highly centered on the customer because in the Agile methodology the customer is part of the development team. The customer role referred to as a *product owner* (PO) works with the team from inception to the end, helping build a set of requirements, prioritizing them, providing regular feedback, and supporting the team every step of the way.

The union of mind-sets puts all other conflicts on the back burner. It does not matter if one is sequential and the other is concurrent because at the heart of both the customer is elevated to a new high. As long as true north points to the customer, frameworks and philosophies blend like fresh strawberries and milk in a strawberry milkshake. The taste gets better only when the individual elements of the milkshake do not reveal itself as strawberries, milk, or sugar. Likewise, ITIL and DevOps have all the ingredients to come together to create synergy and support what customers wants, when they want it, and maintain it seamlessly. The union of mind-sets despite the differences is illustrated in Figure 3-3.

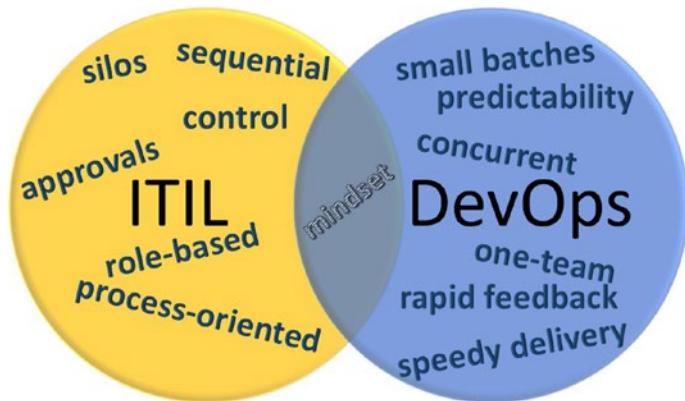


Figure 3-3. Union of mind-sets

The Case for ITIL Adaptation with DevOps

Some experts believe that ITIL is a 20-year-old framework that does not fit into the Agile scheme of things, which is viewed as dynamic, fast-paced, and innovative. Further, the argument goes on to state that ITIL is bulky, rigid, and strictly sequential. The problem these proponents state is that ITIL is a framework for services and service-based organizations. Now with the advent of DevOps, the line between development and support is closing in, and a service-oriented framework ill-fits into a hybrid (development and operations) way of working. Before ITIL, development ran on a waterfall model, and support drew inspiration, practices, and processes from ITIL. With the conglomeration of development and operations, waterfall has already made way for Agile project management methodologies. And ITIL too will have to buckle in. But what is its replacement? There is none because DevOps infuses plenty of enthusiasm in the development practices, and the interest wanes during the operational practices.

The proponents against the ITIL framework are rather biased. Without an occupant in the wings, the existing tenant is being driven out. A big gaping hole is what will be the outcome if their will is the way. Their arguments rather seem one-sided as the obvious benefits derived from having ITIL are often sidelined.

ITIL is a framework that has met the rubber on the road more than any other organized set of practices for more than three decades. The framework has adapted to the changing world and adopted the best practices from the industry to be the be-all and end-all. ITIL v3 came out in 2007, with a minor update four years later (ITIL v4 is expected in 2019, with modern methodologies included). So, can a decade-old framework in the digital age where the way we view and do things has changed

drastically still suit our needs? The answer is yes. Fluidity in development and being Agile does not conflict directly with the principles of ITIL. The way to go about building a service does not change, but what essentially needs to happen is for a doyen in ITIL to identify the pieces of the framework that can be directly inserted into the DevOps processes and to delicately tweak a few more bits without altering the meaning or objectives of service management.

With the absence of any other suitable competitor to ITIL in the digital age, there is no question whether ITIL still needs to be pursued. The only factor is how quickly you are going to recognize the operational needs of DevOps and blend the ITIL framework into the overall scheme of DevOps. Without ITIL, there is no DevOps!

Note Gene Kim, the author of *DevOps Handbook*, says that DevOps being in opposition to ITIL is a misnomer. Even releasing 10,000+ deployments/day requires processes, but what goes against the DevOps objectives are the approvals.

To Conclude

Many experts believe that ITIL may be humongous in terms of the literature and may be sequential, but it is well known that ITIL does not prescribe the sequence of activities nor does it dictate the bureaucracies to be involved. Instead, the framework provides the phases in which a service needs to be developed to ensure that all aspects of a service are addressed during its formational stages.

Jayne Groll, the CEO of the DevOps Institute, says that within the 2,000+ pages of ITIL publications, there are no implications or directions to suggest that the ITIL processes must be developed in a complex manner and that it must follow certain bureaucracies (such as obtaining approvals before embarking onto the next stage). She further goes on to state that DevOps does not diminish the value of ITIL. Instead, it validates and matures it by connecting the dots between Agile, Lean, automation, and other related frameworks.

Matthew Skelton from Skelton Thatcher Consulting believes that DevOps has much to learn from ITIL such as identifying and addressing dependencies, putting emphasis on service level agreements (SLA), and developing service strategies. On the other hand,

ITIL too can learn plenty from DevOps, such as collaboration between various teams, effective event management through modern monitoring toolsets and metrics, and the rapid responses to incidents.

Kaimar Karu, head of ITSM at Axelos, is of the opinion that ITIL has things to adapt from DevOps—mainly on the people front. DevOps can help in better communication, collaboration, and customer focus.

In the remaining chapters of this book, the focus will be on the solutions to the problems and conflicts listed in this chapter. Furthermore, the remainder of the book will take on the integration of the ITIL framework and DevOps practices. I have worked in the area of ITIL for several years and have helped organizations create value through service management designs based on ITIL processes. The ITIL processes are grounded in maturity, and there is nothing fundamentally wrong with them. In the era of swift turnarounds and automation, the existing processes have to be adapted to the ITIL way of working. You will see in the chapters dedicated to individual processes that the principle underlying the processes does not change; the meaning and its objectives stay firm to what it is meant for. But the way it is designed and executed has been altered to make it more dynamic and Agile.

I also like to state that the adaptations that I have recommended are one way of looking through the adaptation lens. There could be a few more ways of doing things. In the spirit of ITIL, where the framework stays away from prescribing a way of designing and operating, I will stick to the same philosophy with my material so you can adapt it further to your organizations, designs, and processes.

CHAPTER 4

Integration: Alignment of Processes

From this chapter on, we start with the exercise of aligning the ITIL service management framework for a DevOps project. The exciting bit starts here! The exercise involves looking at ITIL holistically and identifying the phases and processes that can stay as is and those that need to rejigged.

Although ITIL offers great insights and learnings, I will not reframe the DevOps processes, methodology, or philosophy either to fit the ITIL framework or just so it sounds better.

Analysis of ITIL Phases

ITIL is broadly categorized into five phases, and at least four of the five phases are sequential in nature. The fifth one is an overarching phase. The sequential phases are as follows:

1. Service strategy
2. Service design
3. Service transition
4. Service operations

Continual service improvement is the overarching phase that is invoked at any time during the four phases. Figure 4-1 shows the phases of ITIL.

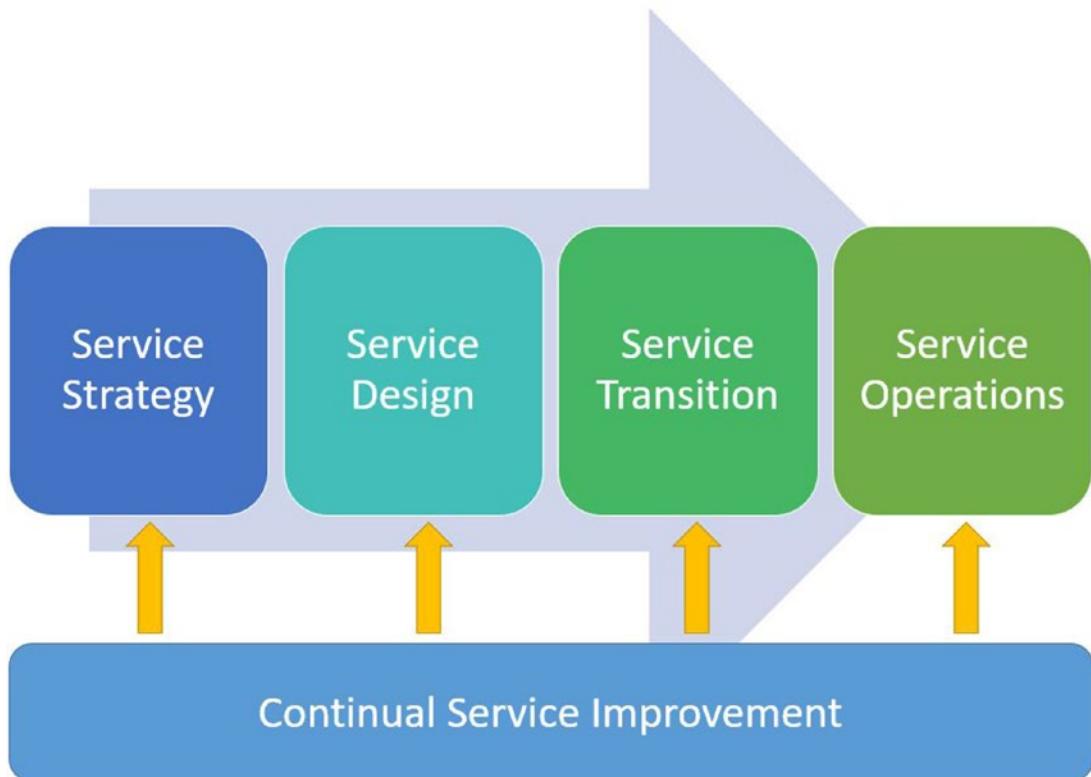


Figure 4-1. ITIL phases featuring sequential and overarching phases

The phases are designed and aligned in a logical manner, where the service ideation and inception happen right at the beginning. The business providing the services break their heads over the service applicability, the cost of its development, the return on investment, and the market segment, among other tasks. This is done in the service strategy phase. After the sponsors put their weight (and money, of course) behind the finalized service, it goes into the design stage. In the service design phase, the various design elements of a service are put together. Here, the service is developed on paper—something like a blueprint. The next logical step is to take this blueprint and get something out of it. This is the service transition phase where the service is built and implemented based on the designs. This could involve training and hand-holding along with the service build. After the implementation of a service, the service needs to be maintained to ensure that it runs as it was designed. This is the service operations phase. Throughout the four phases, there are opportunities for improvement, and the

continual service improvement phase has its stethoscope firmly placed on the four phases, listening in keenly, identifying improvement opportunities, and executing the improvements as feasible.

The obvious problem statement is its sequential nature. Let's say that the business heads sit together and conceive a service takes about a month with the design and transition phases lasting for about six to eight months. Within the nine-month period, a lot can change. The service that was conceived may become irrelevant, or it may have to be overhauled to meet a different market segment. This drawback is typical of a project executed in the waterfall style of project management. The obvious solution is to turn it into the Agile style of functioning where the service elements are designed and executed in sprints. However, adapting service development into an Agile style may not be a simple task.

Analysis: Service Strategy Phase

Let's take another look at the processes in Figure 4-2.



Figure 4-2. Service strategy processes

Strategy Management for IT Services

When it comes to strategizing, there is plenty of research needed to decide what can be deemed a service that can render successfully. It requires plenty of footwork in terms of crunching the numbers and identifying whether it is worth offering a particular service. It also requires identifying the competition and defining a unique factor that will help the organization's service stand out from the crowd. All of these activities and the related ones are carried out under the strategy management for IT service process.

CHAPTER 4 INTEGRATION: ALIGNMENT OF PROCESSES

This process, although associated with IT, is a business process. Every business starting out with a new product or a service has to do the groundwork if it wants to be sure that the product or services will trounce the competition. The flexibility of the Agile way of working is that the end-to-end strategizing need not be done before the rest of the processes kick in. As long as the big-ticket features are known, that is probably sufficient for the high-level design to take place. The same concept can be applied in the deliverables of the strategy management for IT services process.

Let's say that a business wants to offer cloud services as part of its renewed strategy. The market is already crowded with a number of players, and there are some top guns like Azure, Amazon Web Services (AWS), and Google Cloud. The new cloud service must be competitive either in terms of pricing, features, or both to stay relevant. While the basics of a cloud service remain well known, the strategizing is done around the differentiator. For the service design to kick in, is it necessary that the entire strategy is known? Not really. The design can start at a high level, and the basic architecture can be put in place by the time the strategy is well conceived. The differentiators or the enhancements can be on top of the basic design. This technique will help the business to quickly move from the inception toward the design stage and subsequently build and service release.

Note A service backlog is a list of all the basic, intermediate, and advanced features that need to be designed, built, and implemented to deploy a service.

DevOps primarily deals with building (service transition phase) and maintaining (service operations phase) products and services. Therefore, the strategizing and design are taken care of under the Agile project management framework. This is the activity that ensures the product backlog consists of all the elements of the product. In our case, the service backlog consists of all the basic, intermediate, and advanced features that have been identified.

Figure 4-3 illustrates how a service can be built from the ideation phase to the deployment stage in an Agile manner.

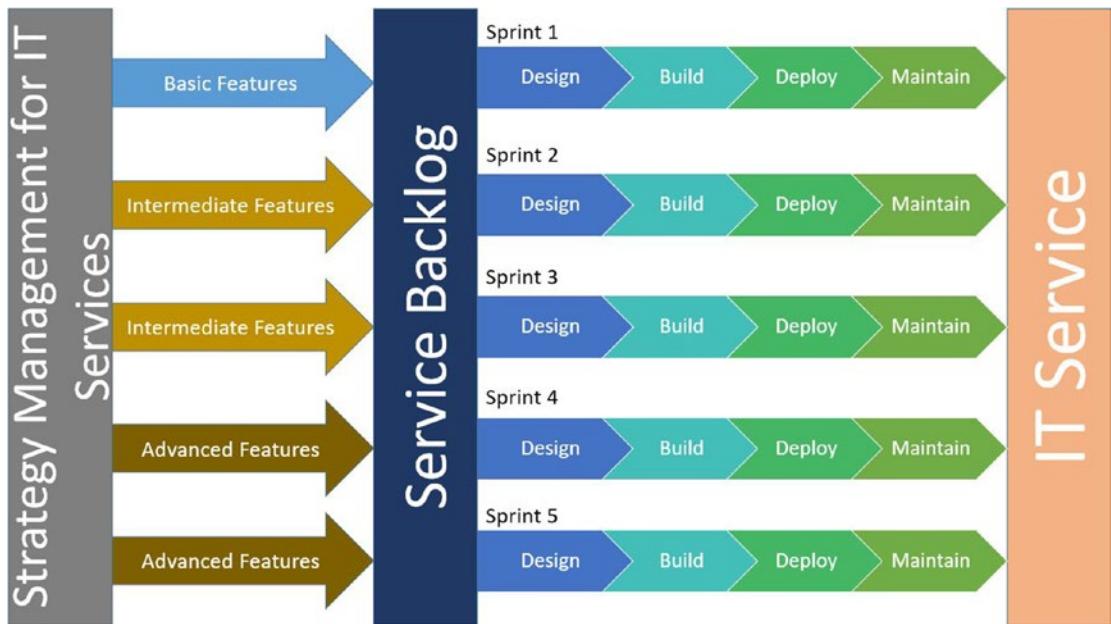


Figure 4-3. Service development in the Agile way

Let's say that the basic features are known early and there is clarity around what goes in, which is usually the case in terms of basic features. An example could be cloud infrastructure to set up servers on the fly using an infrastructure-as-code (IaC) model. The basic features make up the initial items in the service backlog. In sprint 1, the design of the service with the basic features is completed. This is followed by the build (development), and is implemented (deployment). A service can be offered with the basic (but core) features to start with. With the design, development and deployment of the core feature, we have just created the first piece of the IT service. Whether the basic service can be offered to a select group of customers is a business decision.

Likewise, as and when the intermediate and advanced features are formalized, they are populated in the service backlog and subsequently designed, built, and deployed. The service gets developed progressively rather than in a big-bang approach.

After deploying services, it needs to be maintained to ensure the status quo. The operational activities kick in to ensure the smooth running of the service. The activities that you see in Figure 4-4 between the service backlog and IT service usually represent the scope of DevOps.

Service Portfolio Management

The investigation, research, and decisions behind identifying the services to build and offer get done in the service portfolio management process. This process looks at the end-to-end portfolio of services starting from the ideation stage up to retiring services as and when they are no longer relevant and valuable. The process sources the investment into building services and also ensures that the investment bears fruits for the service provider organization.

From a DevOps perspective, this is an activity that happens much before the actual build begins including the design stage. And it is a critical process to ensure that proper due diligence has gone in before committing to it. Therefore, proposing to hasten decision-making through DevOps methods is not feasible.

Financial Management for IT Services

Whether you deal with IT services, IT products, or spare car parts, finances have to be managed. This includes developing a budget to operate within, keeping a tab on where the monies have been spent, and charging back as necessary. At a high level, this is common in most industries, and it is no different in the IT services sector. The financial management for IT services takes care of budgeting, accounting, and chargebacks for services.

The financial side of a service is placed in the service strategy phase; however, this process comes into play in all the phases. While the budgets are set up before the actual work begins, accountability of expenditures happens throughout the lifecycle. Chargebacks too happen as and when necessary. As I said before, this happens everywhere, and it is no different in a DevOps project where high-level budget estimates are made for the build and test phase—because the Agile processes try to couple finances loosely with the project deliverables. From a DevOps standpoint, nothing needs to change the way it is set up in ITIL.

Demand Management

Demand management is an interesting process and not something all that common in most service-based organizations. The process gathers inputs from various sources to feel the pulse for the upcoming demand for services. For example, if the customer plans to add a couple hundred more users to the lot, the service provider must be aware of this ramp-up to ensure that the services such as Internet bandwidths, storage allocations, account creations, and so on, are scaled by the time the ramp-up is done.

So, how is it relevant to a DevOps project? Consider this: as part of the build activities, or even before the build can start, a number of prerequisites need to be in place. How will these demands be addressed without a mechanism that is in alignment with the DevOps ways of working? I know that in most organizations, getting the right kind of people, tools, and especially environments on time is a major challenge. Most of the resources sought are showstoppers, and they make or break organizations. So, how can we align the needs of the project with the demand management process?

The ITIL demand management identifies demand for services through pre-empting and identifying patterns to better serve customers through uninterrupted and optimized service delivery. In a DevOps project, my recommendation is to extrapolate the reach of the demand management process to look into all the demands of a DevOps project. Be it in the human resource area or tools or anything else that is needed to run the project, demand management steps in.

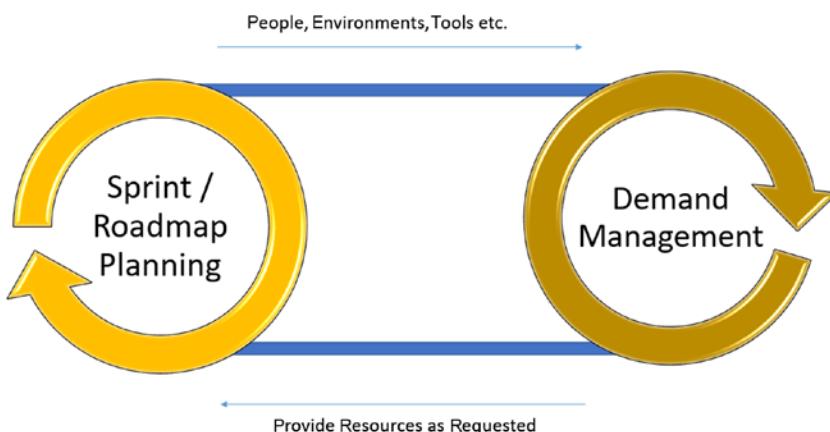


Figure 4-4. Demand management alignment

When we plan the road map for a DevOps-styled implementation, we do certain due diligence to identify what type of people we need, environment setup, tools, work locations, and other work enablers. The demand management process in this setup must be as lean as possible to ensure that provisions are made for speedy deliveries. Today some of the environment and tool setups are managed on the cloud, and the developers and DevOps engineers can themselves spin environments along with the required toolset at the snap of a finger. However, there are some organizations that are skeptical, and the provisioning of environments is still done in the traditional ways. So, if we are going to work in a DevOps project, the demand management process must

make commitments to use the full extent of technology to remove the blockers that may exist. When it comes to people and work locations, it's a different matter altogether. Companies don't maintain a massive bench strength anymore, and brick-and-mortar offices come at a premium. So, there is all the more onus on the demand management process to ensure that the people sourcing and other enabling processes are optimized and focused on speed and quality.

Most of the demands generally get sorted out during the road-map planning. No matter how detailed the plan is, calculations go awry. During the build process, there might be new requirements that need to be sourced. Therefore, there should be a constant alignment between the sprint planning process and demand management to get new demands (if any) every two weeks and fulfill them in a decent amount of time. It should work both ways.

The sprint planning sessions must identify what is needed in the next sprint or a couple of sprints from now, and a governance structure should ensure that the demand management process has open ears to pick up on the demand. A tool such as Slack or Trello will serve nicely in putting up the demands on a regular basis and maybe follow it up with a meeting for clarifications.

Note I don't recommend using collaboration tools for clarifications as it normally goes back and forth, and precious time is spent on clarifying rather than acting on it. It's better to talk and sort it out!

Business Relationship Management

The business relationship management process ensures that sufficient hooks are drawn between the customer and the service provider organization at strategic and tactical levels. This will ensure that the customer feedback is immediately dealt with, and when there is a need for new products and services, the business relationship management can jump in to offer services to meet the needs. This is a relatively new process brought in to support the service level management process in the service design phase.

The tenets of managing customers are more or less the same in every industry, but maybe even more so in a DevOps project. Decisions taken or to be taken at a strategic level can have bearings on the product or service delivered, so it is imperative that an ear is kept to the goings-on of the customer and another on the ground.

Note In a Scrum team setup, the product owner is the customer who works with the DevOps team. He/she provides regular feedback during the various Scrum ceremonies. The gathering of feedback and acting on it is a good example of the business relationship management activities (not all) in action.

Analysis: Service Design Phase

The service design phase comes right after the service has been identified to be developed or enhanced, and the overall design of a service is done here. Making it relevant to a DevOps project, we can consider that this phase provides the architectural and enterprise design to the service or product being delivered in DevOps style.

Figure 4-5 shows the service design processes.



Figure 4-5. Service design processes

Design Coordination

Design coordination is the umbrella process for all design activities. It is a process that manages the complete design activities, including aligning with the strategic initiatives, the business requirements, and the management of budgets, resources, scope, quality, and schedule. To state it simply, design coordination is a process for managing the project of developing service designs.

Most organizations that implement ITIL don't try to call the design coordination process out and plan to meet its objectives. The process just happens under the guise of project management. There is nothing specific about coordinating design in the ITIL framework. However, when we look at designing services for a DevOps project, we can definitely enhance this process to fare better.

For starters, in Agile and DevOps, we like options. We don't like to concretize anything right at the beginning and put all our monies on a single rider. I borrow the concept of coming up with multiple design options from the Scaled Agile Framework (SAFe). SAFe introduced a set-based design (SBD) where the requirements and design options are flexible, even in the development stages. By staying flexible, the design can pivot from one option to another based on the market conditions, the value generated, and expenses, among other factors.

In a traditional world of design, after the requirements are drawn and fully analyzed, the blueprint of the service (design) is drafted, run by all stakeholders, and finalized. This is referred to as a *point-based design*, as depicted in Figure 4-6.

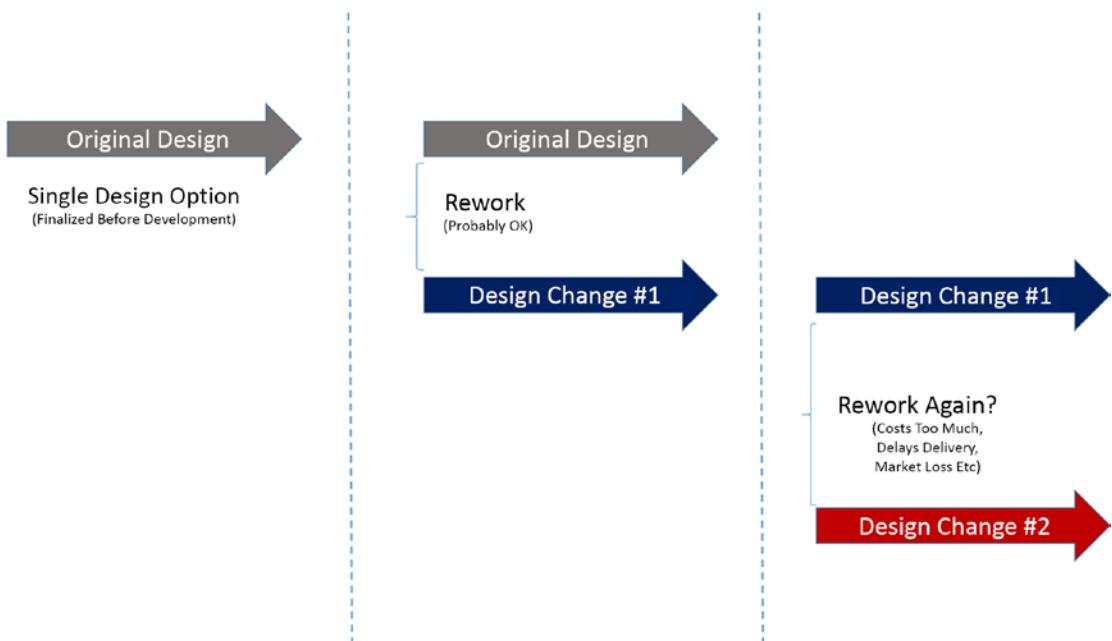


Figure 4-6. Point-based design

Typically, in a point-based design, a single design option is finalized, and the design team gets to work in materializing it, indicated in the left column in Figure 4-6. At some point in the development stages, the original design might require some changes, and these alterations are done on the fly by taking certain approvals. There is a certain amount of rework, but the overall project plan would have factored for such rework, which is still considered normal. In Figure 4-6, the altered design is indicated as design change #1, indicating a decent amount of rework.

However, as the development comes close to ending and as the transition to real world begins, the stakeholders realize that design change #1 is no longer fully relevant to the current market conditions and to what they had assumed a few months/years back. The design needs to change, and that needs to be followed up with a major amount of rework. These are questions that pop up at this moment: Is it really worth it? What if something were to change between now and the development completion of design change #2? What is the return on investment from all this rework?

There is a good chance that the service that needs to go through massive amounts of rework might never see the light of day.

What if there were multiple designs to work with from the beginning? What if we could pivot during the development stages from one design to another? The set-based design has the answer (see Figure 4-7).

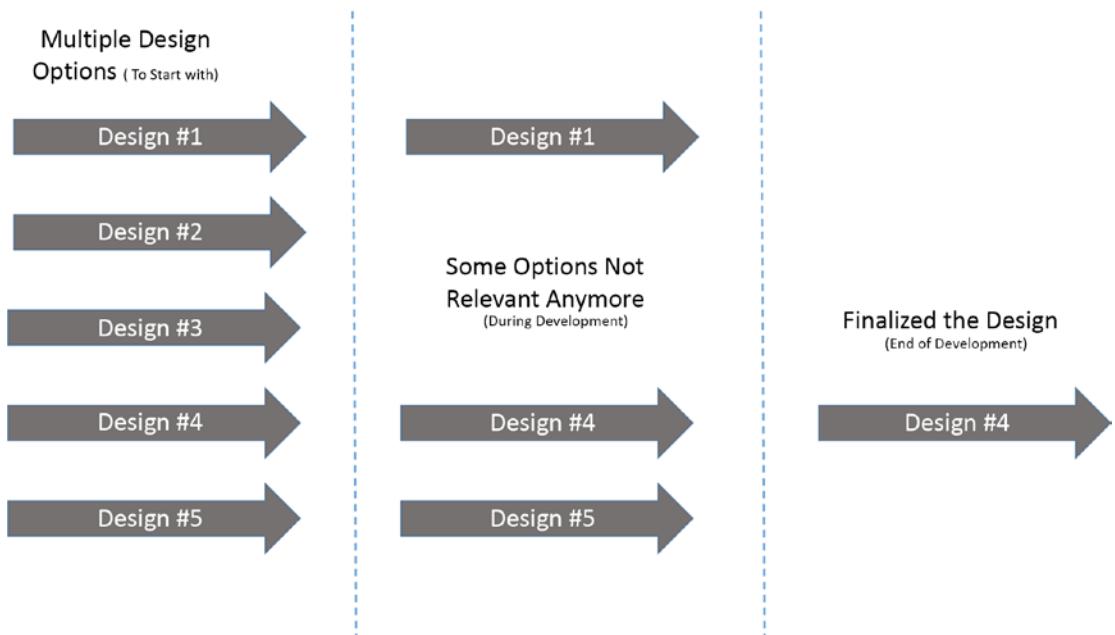


Figure 4-7. Set-based design

CHAPTER 4 INTEGRATION: ALIGNMENT OF PROCESSES

In set-based design, we don't finalize on a single design. We come up with a set of possible designs, which gives us the leverage to pivot from one design to another, even late in the game. This is illustrated in the first column where we finalize on five different designs.

The development begins with all five designs. When we do this, we start with aspects of design, which helps when making decisions to drop some design options. In Figure 4-7, we found out that designs #2 and #3 are no longer relevant.

We continue working on the other designs, and as we come closer to completion, it is clear that some designs can't make the cut, and we finalize design #4.

In set-based design, instead of finalizing at the beginning and working our way making changes to the design, we start with multiple options and eliminate designs that don't fit the purpose and the use. This process gives us a better handle on adapting to the current market conditions faster and to make informed decisions as we move through the process. Most importantly, we don't have to live with a mistake that we made months earlier in deciding on a design over the other but rather make a decision when we have all the data points available to us.

Now let's address the elephant in the room: rework. Working on multiple designs just to eliminate them during the process is wastage of time, resources, and money, isn't it? Not really. Think about it, the development that goes against the designs are all planned and does not come into the category of rework where you try to alter the existing design to fit the updated set of requirements. Moreover, the whole objective of developing services is to ensure that it creates value to the customer, and to achieve this goal, if it requires additional effort to generate extra data points through multiple designs, so be it!

Another option is to borrow the principle of minimum viable product (MVP) from the Lean Startup mind-set, where the design considers the bare minimum aspects of design to get started. When the minimal design is put into development, the outcome will provide the necessary data points to make informed decisions.

The goal here is to develop a design that fits the purpose and is fit for use for the current market conditions, at a finite time and at a planned budget. The percentage of success is much higher when there are options, and options give us the flexibility to alter our course as the wind blows.

Service Catalog Management

The service catalog management is a key process in the ITIL framework. It provides a single view of all available services for the customer to choose from. Think about the service catalog as a menu in a restaurant that gives you a list of all available dishes. Likewise, a service catalog for a service provider gives customers the necessary information on what is being offered and also what is in the pipeline.

Service catalog management exists to ensure that the service catalog is current and up-to-date and provides all the pertinent information to the customer. This works well in the ITIL framework and can continue to do so in a DevOps project.

There is a second thread to service catalog management, which is more technical in nature. It is referred to as a *technical service catalog*. This is an internal service catalog that provides the internal teams with pertinent information on who is supporting the service and what the dependencies are on one another.

For example, if I was to offer WordPress as a service to the customer, on the back end, multiple teams make up this service. The cloud infrastructure team takes care of the underlying servers and networks. The application team provides support for the WordPress application. The database team supports the MySQL database. All three teams work in conjunction to provide the WordPress service. If any of the underlying services that power the WordPress service go down, the WordPress service goes down with it as well. Therefore, it is important to map the service dependencies, and the technical service catalog maps the service dependencies and keeps it current and updated.

In a DevOps project, technical service catalogs are as important as the service catalog (visible to customers). Based on its availability and accuracy, DevOps can be a lot more effective by churning out incident fixes, identifying dependencies, and mapping components and modules. In a typical DevOps project, the role of a technical service catalog is often not recognized, which leads to delays owing from analysis to identify the dependencies. A DevOps project can be a whole lot stronger and effective with the implementation of the service catalog management process.

The service asset and configuration management (SACM) process is an extension of the technical service catalog management process, and it maps the dependencies to its granularities. I have dedicated an entire chapter to this process (Chapter 6).

Service Level Management

The service level management process is like a score-keeper. It is responsible for measuring the levels of services delivered to customers, and before it is able to measure, the process agrees with the customers on the level of service that is to be offered.

The service level management process works closely with the business relationship management that we discussed in the “Analysis: Service Strategy Phase” section earlier in this chapter. While the business relationship management works at a strategic and tactical level, the service level management works at an operational level.

This process seeks to understand the service level requirements (SLRs) from the customer and translate them into service level agreements (SLAs), which will be used as a rulebook for measuring and reporting the numbers. The SLRs determine the contingencies to be provided in the architecture (such as high-availability architecture) and the resources to be onboarded, and this eventually has a bearing on the cost of services.

The service level management process under ITIL was meant to measure the various aspects of a service, mostly on the operational front. DevOps is a combination of development and operations. While the operational measurements hold water in DevOps, the development measurements need to be factored in during customer discussions and negotiations. Generally, in a DevOps project, discussions around key performance indicators (KPIs) and expected measurements are done at a contractual level without a process defining the ins and outs of the data sources and measurement of performances. The service level management can be a valuable ally for DevOps projects in identifying the KPIs and keeping a tab on how things move along.

Agreement of service levels happens at multiple levels. The agreement with customers are referred to as SLAs, and an internal agreement within the same organization is called as an *operational level agreement* (OLA). Typically agreements with suppliers too have SLAs attached along with underlying contracts (UCs).

DevOps is all about speed and quality. To enable speed, multiple toolsets are employed across various environments. They are usually managed by separate teams. Therefore, it is imperative that the agreements (either OLA or SLA) exist to ensure guarantees for speedy deliveries.

Availability Management

The availability management process ensures that the services delivered to customers are available as mandated by the service level agreements. The premise is that no matter how great the offered service is, it is not worth anything if the users are unable to access the service. This process ensures that the underlying architecture—the infrastructure and application—are built to the expected rigors of the service.

After the service levels are agreed on by the service level management, the availability management kicks in to build a service that meets and perhaps surpasses the expected levels of a service. For example, building a service for 99.99 percent availability can look very different for a similar service built for 99.9999 percent availability. The difference between the two is minuscule, running in decimals, yet the impact on architecture could be massive. For 99.9999 percent availability, the architects have to factor in multiple layers of contingencies to ensure that even multiple failures do not take down the service. The cost of services offered for 99.9999 percent availability is in multiples of the cost of services offered for 99.99 percent availability.

Managing availability in an ITIL project is similar to that of a DevOps project. Both offer services, and both have to abide by the service levels pertaining to availability. However, there are additions as always in the DevOps world. First, most environments are on public clouds, and this could be a challenge in terms of setting availability targets. Then, the presence of multiple environments to provision for builds and tests requires a tightened grip over the availability designs, even for environments that are not customer-facing. Also, there are multiple tools that are employed in DevOps projects, and they too need to be hosted, and they too have to be available as per the DevOps team's needs. Any laxity in the availability management of test environments or tools will result in the delayed delivery of services, which in turn will affect the overall delivery levels set forth by the service level management process.

Therefore, it is important that the availability requirements are carefully analyzed to be in perfect alignment with the delivery rate and the speed at which the team is able to deliver.

The other aspect to consider is over-delivering on availability. As I mentioned about the cost earlier, every single decimal adds exponentially to the cost of services. So, building an architecture that over-delivers in terms of availability is not bound to be cost-effective, which is detrimental to the business angle of services.

Capacity Management

Although capacity management is placed in the service design phase in the ITIL framework, the process extends its arms across the entire lifecycle. It exists to ensure that the services offered have sufficient capacity to produce value. As a service provider, you might be offering a top-of-the-line, full-featured service (say cloud services), but if there isn't sufficient capacity (bandwidth and disk space), then the service will render useless after all. Therefore, capacities must be ensured for maximum utilization of a service, and the process plays a significant role in value creation for customers.

Note In the traditional architecture design principles, the design will also leave room for growth and uncertainty; however, in the DevOps world, the demand might vary because of changing business requirements and the shortened service/product lifecycle.

The capacity management works proactively in planning for various aspects of capacities and also in reacting to capacity-related incidents and problems. It is a process that takes control over anything to do with capacity across the entire lifecycle of a service or a product.

Capacity management comes in three flavors.

- Business capacity management
- Service capacity management
- Component capacity management

Business Capacity Management

The business capacity management flavor tends to the business's needs for ensuring sufficient capacities. It works primarily in the service strategy phase and is responsible for understanding the demands of the customer, and it works closely with the demand management process in identifying patterns of business activity to accurately plan for current and future capacities.

In a DevOps project, business capacity management plays a key role in ensuring that the scale-up and scale-down of deliveries are done in a seamless fashion by monitoring the demand and supporting the scale plans.

The process can also help provide guidance on the scale of workload coming the project's way. Think about the incoming requirements as a funnel. The customer throws in a bucket a list of the things to be brought forward to production. The funnel size determines the incoming flow and the outcome. Therefore, insights around what comes in can help alter the funnel size as needed. On the flipside, a scale-down insight will help optimize and reduce the overall cost of delivery. Figure 4-8 illustrates the funnel concept.

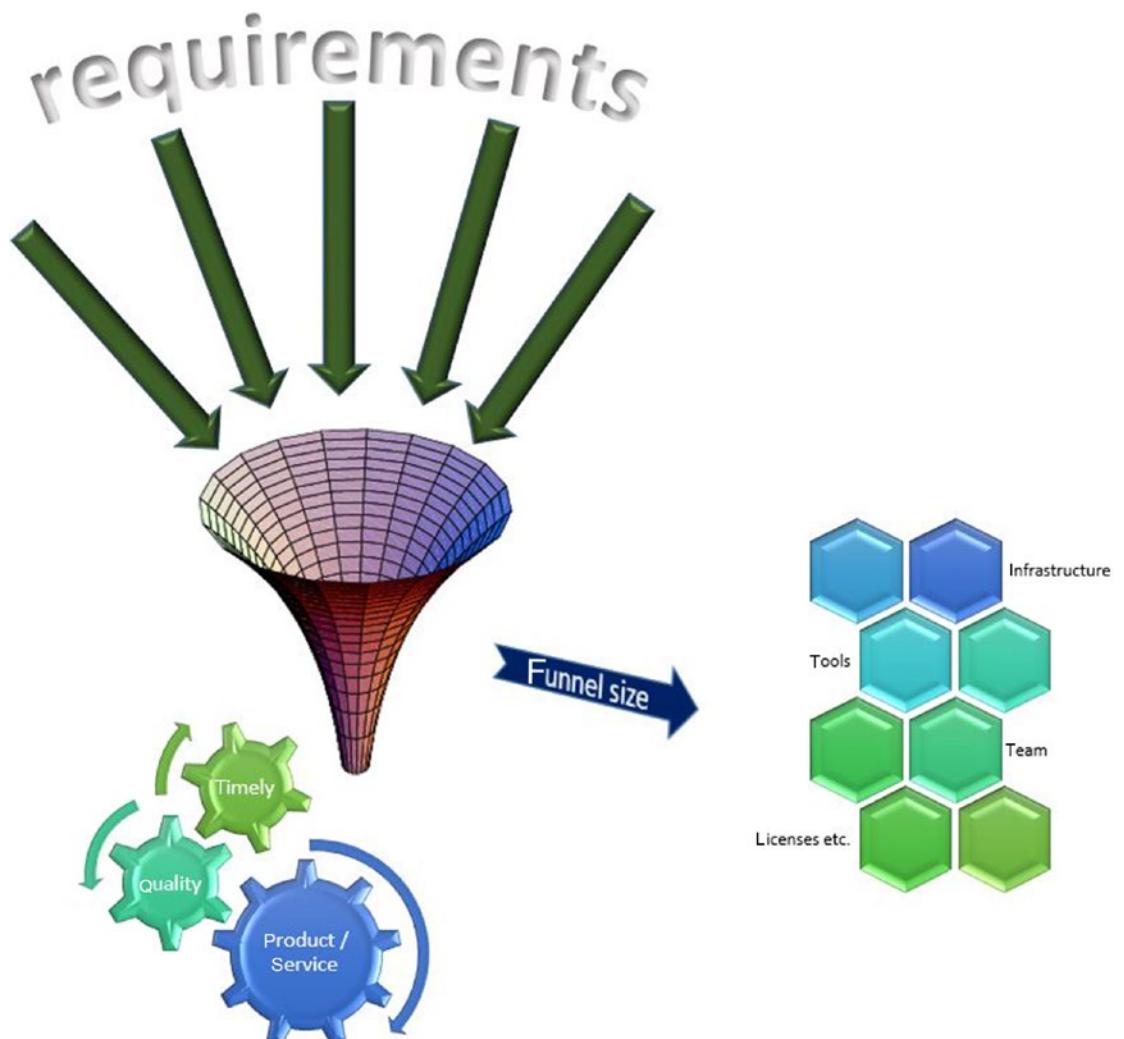


Figure 4-8. Business capacity management

In the figure, requirements come thick and fast through the opening. The service or product delivered will depend on the size of the funnel, which is indicative of the infrastructure, tools, team strength, and licenses, among other things, that are required to deliver. The delivery is a product or a service, and without any riders, a product or a service does not add any value. What if a service is delivered that's reeking of bugs or a product gets delivered two years too late? It is critical to deliver when the product is expected to be delivered, and an agreed level of quality must be maintained as a hygiene factor. All this is possible if and only if the funnel size alterations are made quickly and effectively. For it to happen, business capacity management is a vital cog in the entire chain link.

The ITIL business capacity management is well-defined to address the capacity concerns coming from the business. For DevOps, the existing process can be readily applied without any modifications or enhancements. As it is deployed in the cloud technology, we can ensure the capacity can be increased anytime you need it or decrease if demand drops to a certain level. The elasticity of capacity demand is suitable to deploy on a cloud-based platform.

Service Capacity Management

The service capacity management operates one rung lower than the business capacity management: at a tactical level. The process delves deeper into the services offered and brings to the table an intimate understanding of it. Through service capacity management, we will understand the resources that are leveraged for the delivery of services, the usage patterns of the service, and all the other statistics. Having this information powers the service provider to maintain the performance at the required levels and to optimize service elements wherever necessary.

For example, a video streaming service provider can improve their content delivery networks (CDN) in regions where the subscription rate is high and optimize resource at regions with moderate subscribers. The only way to make decisions on enhancing performance or optimizing resources is through service capacity management.

The service capacity management process is relevant in its present form for DevOps projects as the methodology can leverage the maturity of the process to keep its ears close to the ground from a capacity perspective and make adjustments in an Agile manner as and when necessary.

Component Capacity Management

The final subprocess that operates at the runway level (operations) is the component capacity management. In this subprocess, the scope isn't to measure the performance of the service from end to end but rather look internally at the various configuration items that make up the service.

Component capacity management is an important subprocess as the tactical and strategic capacity management processes rely on the data from the ground to make decisions on the top.

In a DevOps project, where a service is dependable on the infrastructure and applications, every single cog being capacity managed is a given. The additional components that need to be capacity managed are the tools that are employed and the various nonproduction environments that are set up to manage the quality show.

IT Service Continuity Management

The IT service continuity management (ITSCM) process supports the business continuity management (BCM) process by ensuring that the business functions identified in the BCM are sufficiently recovered in the agreed upon timelines.

This process is invoked mostly when there are disasters of epic proportions and the service offered to customers is probably going to be impacted for long periods of time. Long-running incidents without any resolution in sight can fall under the scope of ITSCM too.

The service provider would have pre-arranged recovery options such as having real-time data replication on servers sitting across the ocean or running empty fully equipped offices to helicopter in people from affected regions as deemed necessary. There are different types of recoveries, but what is more important is to identify the type of recovery needed and to make plans for it. The inspiration to identify the type of recovery can be drawn from the BCM.

The ITIL ITSCM process is meant for services that are offered and used by customers and deemed critical to customer's business. The process framework is mature and has demonstrated time and time again that it meets the objective and is valuable to the customer. So, I see the ITIL ITSCM process delivering the same sets of values in a DevOps project for services that are under the purview. On the development side of DevOps, the code, database, and tool configurations are most critical, and if there are backup and recovery mechanisms in place, that should be enough for most projects.

Some projects might insist on the development and testing not to stop even in the wake up of disaster given the stringent timelines running against the market conditions. In such cases, the DevOps architects must plan for recoveries of code, data, pipelines, and environments, and provisions for developers to work in alternate settings must be explored such as working over the raw Internet from the comfort of their homes. Nowadays, the job of architects is less challenging, as the cloud platform can offer lots of features for fast recoveries, backup, and dual-site hosting for disaster recovery operations.

Information Security Management

Information security is a key aspect of IT today, and everything that we do in anything digital is looked at from the perspective of security. To manage the customer information that is used in delivering the services, the information security management process is in place.

The information security management follows the footsteps of the overall business, the customer's business security controls, and legislative controls if any. In ITIL, information security is comprised of the following:

- *Confidentiality*: Confidentiality is about ensuring that only the authorized personnel have the right to access data. It is about protecting information from disclosure from unauthorized parties.
- *Integrity*: Data that is accurate, trustworthy, and consistent defines integrity. Data gets transmitted at some point in the lifecycle and gets read and modified across the lifecycle. The data must stay true to what was transmitted at all times. In essence, integrity is about protecting the data against change from unauthorized parties.
- *Availability*: Data may be well-protected from access by unauthorized people, but it must also be available to authorized parties, as and when needed. Denial of access to data is one of the security concerns and is categorized as a data breach. The service provider must guarantee availability by ensuring that infrastructure is well-architected, and the security protocols protect the right to privacy and safeguard against modifying confidential data.

This is generally referred to as CIA and is illustrated in Figure 4-9.

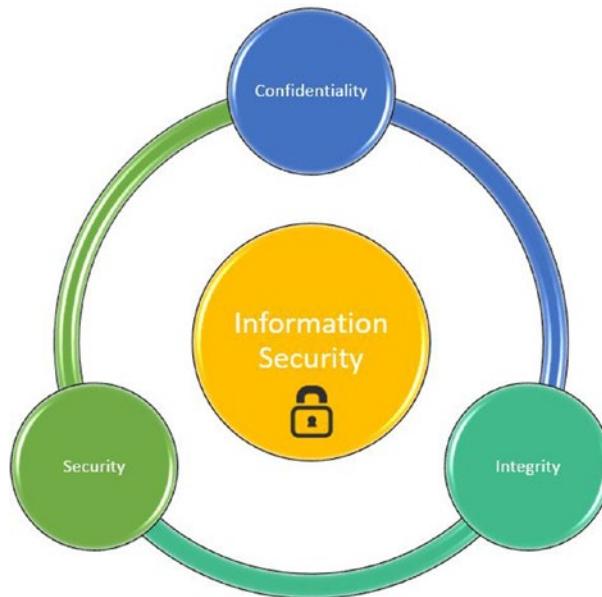


Figure 4-9. Information security elements (CIA)

From a DevOps perspective, CIA is too basic for service operations. From a development standpoint, security is given a lot more standing, and there are two philosophies currently that are ruling the security space in DevOps projects: DevSecOps and Rugged DevOps.

DevSecOps

The philosophy behind DevSecOps is that everyone involved in development is responsible for security. This includes the businesses, developers, tools administrators, testers, and product owners, among others.

Traditionally, security was considered after the product was developed and functionally tested. It was a phase in the development process, and this was in the waterfall world. In Agile, security activities must be done in an iterative manner. But, the problem is that security has always been considered as a major roadblock for speedy delivery and IT innovation. So, in many cases, security becomes an afterthought. There are various instances of security lapses like in Twitter and Facebook, which are a result of laidback security controls.

In DevSecOps, the goal is to introduce security at every single stage of the development process. It starts with the business processes, where the business operators are given the tools and techniques, along with security practitioners, to help in identifying security requirements. This is further drilled down with security practitioners being part of the DevOps team and helping at every step in monitoring the system, attacking the system, and identifying security defects before hackers do.

The thinking behind DevSecOps is that value is truly not added until security has been part of it. It's a mind-set to inculcate and is built on the back of cooperation between various stakeholders. Just as testing has shifted left, security too has moved with it. Now security activities such as identity and access management, vulnerability scanning, and firewalls can be performed programmatically (security as code) and automated.

Rugged DevOps

While DevSecOps ensures that security is considered from the beginning stages of development, Rugged DevOps put the onus on security over all other activities. In other words, between development, operations, and security, Rugged DevOps prioritizes security over the other two.

In Rugged DevOps, the security framework is first established, and all the other successive and subsequent activities will work within the boundaries of security. This promotes an increase in trust, transparency, and a better understanding of risk probabilities.

The objective is to get solid, secure code, and the Rugged DevOps mind-set places stringent controls to achieve this. One of the practices is to perform a penetration test at every stage of development.

The Rugged DevOps Manifesto is as follows.

RUGGED DEVOPS MANIFESTO

“I am rugged because I refuse to be a source of vulnerability or weakness.”

“I am rugged because I assure my code will support its mission.”

“I recognize that my code will be attacked by talented and persistent adversaries who threaten our physical, economic, and national security.”

Supplier Management

No single service provider can provide end-to-end services for a customer. The service provider will employ other service providers to provision services for a customer. For example, a cloud service provider will require dedicated network connectivity, which is a specialty of different types of service providers. The cloud service provider will require hardware, and there are hardware manufacturers that indirectly help in the provisioning of services. The network service providers and hardware manufacturers are referred to as *suppliers* in ITIL as they are not directly contracted by the customer but rather by the service provider to offer a service. Suppliers are managed with the supplier management process. The trend today is adopting another framework that's an offshoot of ITIL called *service integration and management* (SIAM). This framework deals with the management of services in a multivendor environment that is common for most organizations today.

One of the main objectives of the supplier management process is to get most out of the money paid to suppliers. However, in project management circles, we often ensure that suppliers don't get squeezed, and in the end, the contractual situation becomes favorable to the both of us. From a DevOps perspective, we employ a number of suppliers, and the money expended is not everything. We need to build a healthy relationship with suppliers to influence their feature pipeline. For example, let's say that a configuration management tool that we use requires an additional feature such as self-healing with some added controls. If our relationship with the supplier is healthy, we stand a chance in prioritizing the supplier's feature development list, which will eventually be beneficial to us. To sum up, in any project, especially with a DevOps philosophy, it is paramount to maintain relationships, promote collaboration (even with suppliers), and keep an open communication channel with all involved parties.

Other supplier management objectives are about managing suppliers through contracts and measuring performance on a regular basis. These are what you would expect from a supplier management process, and the ITIL supplier management process ticks all the boxes.

Analysis: Service Transition Phase

In the service design phase, all the aspects of design are carried out. In the DevOps adoption, however, some of the processes that appeared in design are carried out throughout the development lifecycle. Considering that the high-level designs are in place, it's time that the rubber starts to meet the road. This is the service transition phase where the actual development of services begins.

The processes involved in the service transition phase are indicated in Figure 4-10. Some of the processes have dedicated chapters for their adoption for a DevOps project.



Figure 4-10. Service transition processes

Transition Planning and Support

Transition planning and support is the project management side of the entire service transitioning activities. The process ensures that all the deliverables promised in the service transition service lifecycle are delivered as per the scope, the time, and the cost. This process is quite specific to the service transition phase. In DevOps projects, the service transition and service operations phases work hand in hand, and the process and specific project management side of transitions will make way for Agile to step in and take over the entire project lifecycle.

In effect, the transition planning and support will cease to exist in its present form and will become part of the entire Agile process that primarily defines and dictates the transitioning activities.

Change Management

Chapter 9 is dedicated to this process.

Service Asset and Configuration Management

Chapter 6 is dedicated to this process.

Release and Deployment Management

Chapter 10 is dedicated to this process.

Service Validation and Testing

A service or product that is built needs to be tested to assure the quality of it. This includes functional tests to prove that the service or product is fit for use and nonfunctional tests (such as performance and security) to prove that the service is fit for purpose. These tests are within the scope of the service validation and testing process in ITIL.

In the ITIL publication, value creation is defined as a product that is fit for purpose and fit for use. Value is derived based on the functionality delivered through a service and the warranty aspects such as nonfunctional parameters such as security, capacity, availability, and continuity. Figure 4-11 defines the value creation principle.

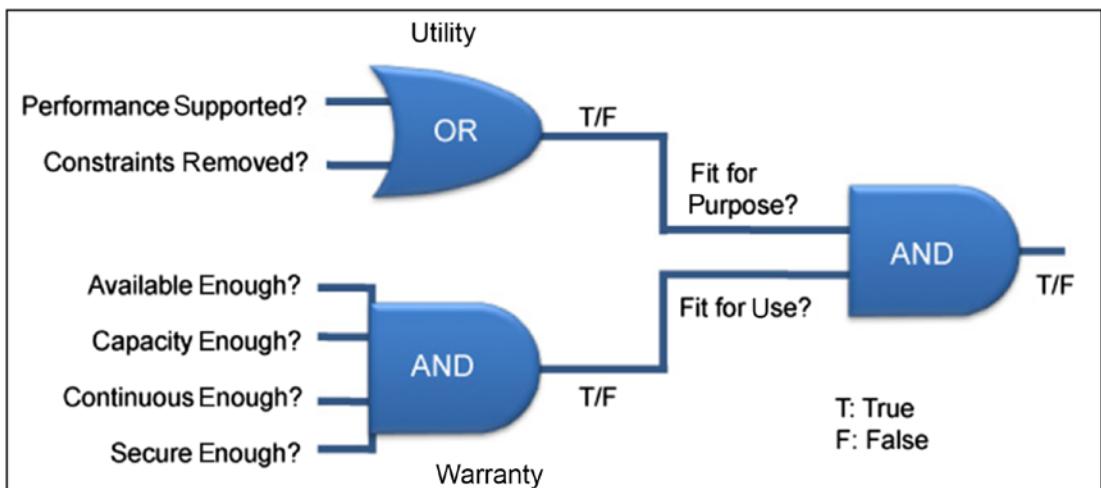


Figure 4-11. Value creation (image credit: ITIL.org)

CHAPTER 4 INTEGRATION: ALIGNMENT OF PROCESSES

This aspect of value creation in DevOps is the continuous testing that is performed after the binary is built. I briefly addressed the testing activities in Chapter 1. Testing ensures that the product or service is in conformance with requirements and it meets all the nonfunctional requirements as well. The DevOps testing process (continuous testing) fits like a hand in glove with the ITIL value creation, as illustrated in Figure 4-12.

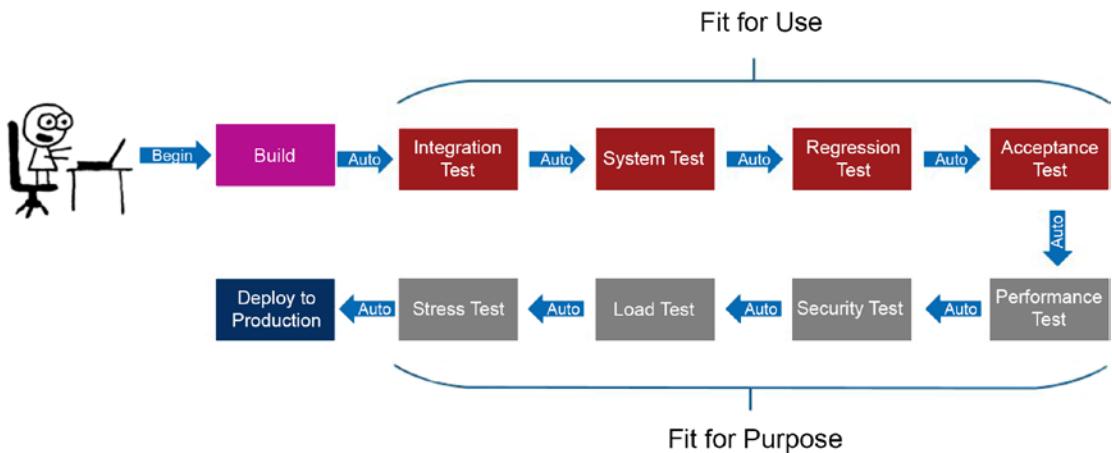


Figure 4-12. Value creation through continuous testing

In the figure, tests such as integration, system, regression, and acceptance are functional in nature, and in ITIL terminology, a test ensures that the service or product is fit for use. The tests indicated in the bottom row such as performance, security, load, and stress are conducted based on nonfunctional requirements and are ensuring that the product or service is fit for purpose.

The synergy between ITIL and DevOps in terms of the existing processes and testing activities is further strengthened by the automation capabilities introduced in DevOps through the continuous testing model. Every single test is done automatically and multiple times, which identifies bugs and assures quality; assuring quality is one of the prime objectives of the service validation and testing process.

Change Evaluation

The change evaluation process is a supporting process to the change management process. It ensures that all aspects of risks are considered and put forth in front of stakeholders who are evaluating a change (in other words, the change advisory board [CAB]).

The process is also responsible for conducting a business impact analysis to identify the true nature of business impact based on the potential change, which once again provides the ammunition to the CAB to make a decision.

While carrying out business impact analysis and conducting thorough investigations are good practices, they must not come in the way of speedy delivery and innovation. In DevOps, the changes made are smaller in size and are done multiple times to arrest the probability of business impact due to a change going south. In other words, changes are done much more rapidly, and if a change brings about an unwanted situation to the business, it will be rolled back immediately. Since these changes are so small, the chances of one hurting the business is minimal. This is one of the insurances that exists in DevOps for delivering rapidly. In essence, the process around thorough investigation may not happen in depth, but it needs to be done for every single user story that is developed.

The concept of minimum viable product (MVP) is borrowed in most DevOps projects. This concept comes from the Lean Startup methodology. In an MVP approach, we first build a service or a product that is as basic as it can be but provides the opportunity to conduct tests such as business impact analysis and various forecasts to get a feel for how the final product might impact the business, positively and negatively.

So, in a DevOps project, change evaluation is not done exclusively but will be embedded within the change and release management processes.

Knowledge Management

Knowledge is most valuable in all areas of study, be it ITIL, DevOps, or civil engineering. It must be built, protected, and managed to have a complete command over of products and services and to make sound decisions. The knowledge management process in ITIL is defined with a sole purpose of managing the knowledge within organizations.

A product or a service gets built over a period of time and goes through multiple iterations of changes. The people responsible for the development and testing change over time and are replaced by others. The incumbent knowledge with the team is precious; it helps them make changes efficiently and resolve incidents effectively. But if a new team is tasked to do the same, the knowledge management system is the only bridge that can scale up the new team to the incumbent team's familiarity with the products and services.

The objective of ITIL knowledge management process is to ensure that the knowledge of services is preserved in a knowledge management database and is updated as and when changes are made. It also attempts to prevent reinventing the wheel because of an effective knowledge management system.

The process ticks all the boxes in the list of features and activities that you look for in a knowledge management system. It is ample and has demonstrated that it works well with the ITIL framework, and the same process can be implemented for a DevOps project as well. However, there is a subtle difference. Agile projects encourage less documentation than traditional projects. The DevOps team uses a less formal way to maintain the knowledge by coding in a way that tells the story, plus writing comments in the code (to make the code more readable), sharing knowledge in wiki pages or user community forums, or even just taking screenshots and videos to capture knowledge. It can be quite challenging to search and locate the knowledge in the DevOps world, as it is not as straightforward as is the case with traditional projects where everything is documented and a keyword search can reveal the results.

Analysis: Service Operation Phase

The service transition phase deploys the product or service into production, and now it is up to the service operation phase to maintain the status quo.

Figure 4-13 shows the processes in the service operation phase.



Figure 4-13. Service operation processes

Event Management

The event management process monitors various strategic points in the infrastructure, applications, and services and keeps a close watch on events that are preprogrammed. Only through event management can a fast resolution be put in place, which reduces the downtime and increases value to the customer.

The process depends heavily on tools to monitor devices and services. DevOps deals with a lot of tools, including tools that fall into the monitoring space. The ITIL event management process provides guidance on the process to identify the critical points, monitoring controls and the subsequent set of actions to be done based on the event. All this can be readily consumed by a DevOps project without a blink of an eye as there is no dedicated guidance in DevOps to support the setup of monitoring systems and event detection mechanisms.

The ITIL event management process can be applied to the DevOps toolsets as well to ensure that the rapid delivery mechanisms don't get affected because of failing tools and infrastructure. It is also key to establish types of events to help understand the criticality of the events generated. Talking of types of events, ITIL defines three types of events

- *Informational event:* This event when generated conveys information that is mostly transactional such as an administrator logging into a server or a completion of a batch job. The outcome of an informational event will not have any subsequent actions attached to it. At the most, an e-mail could be sent out to certain stakeholders.
- *Warning event:* A warning event indicates that something is unusual; however, it is not an exception or an error yet. As the event name indicates, it gives a heads-up that an anomaly is about to happen. Examples include the CPU utilization nearing peak load and an administrator password keyed in incorrectly multiple times. The subsequent action after an event can be a low-priority incident logged and automatically assigned to the designated team.
- *Exception event:* When something goes down or is untoward, it is an exception. Urgent action impends after an exception event. Examples include cloud services that are down and data that is downloaded to an unknown IP. Following an exception event, a high-priority or a critical priority incident is raised, and the resolution teams spring into action.

There is no rule that events must be categorized in this way. ITIL is based on good practices and based on the experiences of multiple organizations. These three types of events have served well over the years; hence, they find a place in the book. Plus, the examples that I have provided are based on my experience and what I think must be

categorized under information, warning, and exception. For your organization, you can start defining events on a blank slate. You can define as many types of events as you want—as long as there is a clear demarcation between the types of events.

Note Monitoring and event management might be talked about in the same vein, but there is a subtle difference. Monitoring is the activity pertaining to keeping a close watch on the device statuses. And event management deals mostly with the aftermath of an event detection. It deals with identifying meaningful notifications and taking appropriate actions.

Incident Management

Chapter [7](#) is dedicated to this process.

Request Fulfillment

The request fulfillment process is defined to fulfill the service requests placed by the users, customers, and other concerned stakeholders. This process is often confused or is badly combined with the incident management process. The request fulfillment process serves service requests, and the incident management process serves incidents. Service requests and incidents are different beasts altogether.

An incident is a disruption to a service, either in full or partially. Generally, it pertains to downtimes and outages. A service request, on the other hand, has nothing to do with outages. It is a request placed to obtain something over and above the service that is already offered. Examples of service requests include access requests to a SharePoint portal, requests for a new laptop, and requests to unblock a certain IP from a firewall.

In a DevOps project, there are plenty of tools, databases, systems, and repositories. Projects may require new tools, upgraded environments, access to team members, and setup of new configurations. All these fall under service requests and fall under the ITIL request fulfillment process. The process has matured over the years, and in the present form, it is apt for a DevOps project.

Problem Management

Chapter 8 is dedicated to this process.

Access Management

The access management is an offshoot of the information security management and request fulfillment processes. While accesses to tools, systems, and repositories are considered to fall under the confidentiality clause of the CIA, access management is also a service request that is managed through the request fulfillment process.

The access management process exists to execute the policies set forth in the information security management process, and it diligently acts on service requests in providing, modifying, and removing accesses to users and other stakeholders.

In my view, the access management process is a minor process that could have been done away with in the ITIL 2011 publication. It does not really add significant value to the ITIL framework, and the activities stated in the process are duplicates of what is already defined under the request fulfillment process. At best, access management could have been a subset of the request fulfillment process. I have not seen many organizations implement access management separately, and if an organization feels a need to do so for a DevOps project, so be it.

Continual Service Improvement

The continual service improvement (CSI) lifecycle phase stretches across all other phases. The scope of CSI is any of the other activities in the other four phases. The objective is to ensure that the services don't stay stagnant but rather improve gradually and steadily. There is just one process in this phase and is indicated in Figure 4-14.



Figure 4-14. CSI process

The Seven-Step Improvement Process

The seven-step improvement process is the only process in the continual service improvement service lifecycle phase. This is so because the process is quite generic to be applied to any situation, any process, and any activity to come out on the other side with an improvement opportunity.

It is important to note that improvements are an integral part of ITIL. If a service stays stagnant without anything to show in terms of improvements, it is pretty much guaranteed that the service is on a decline. A competitor is likely to come up with something better, and the service that lacks improvements is due to be left playing catch-up. So, without any doubt, improvements are a necessity in ITIL. And all improvements, either directly or indirectly, point to an increase in value delivered to the customer, and this and only this will keep the service afloat.

Figure 4-15 shows the seven-step improvement process.

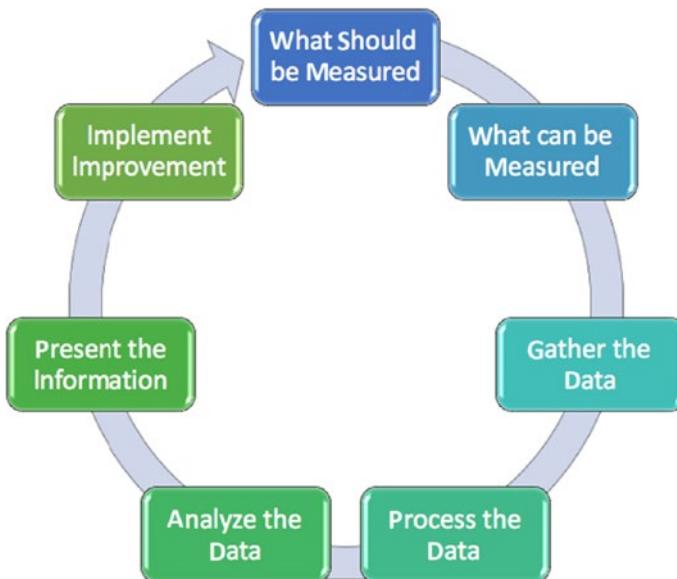


Figure 4-15. Seven-step improvement process

The seven-step process follows a logical model where the succeeding activity builds on top of the first, and all the seven actions are aimed at identifying and implementing improvements. It all starts with what the strategy is, in other words, what we are trying to improve. This points directly to the IT vision that has been set forth. Once we know where we want to be, then in the second step, we check whether it is feasible to find that

data points that we need. If you are unable to measure something, then probably you have an improvement right there—go back to the design table and enable measuring the data points as needed to achieve the targets.

Steps 3, 4, and 5 assume that we have the data that we need, and the data is processed and analyzed so that the mere raw data ends up being information that can be used by step 6, where the analysis is presented back to the decision-makers. When a decision is made to improve, the final step is to go ahead and implement it. The key here is that the process does not stop with one iteration. It keeps chugging along. This model of continuous cycles of improvements is an indispensable factor when we consider alignment with DevOps. In DevOps, we love iterations as it makes sense to keep doing small things over and over again to achieve big targets. Second, improvements are the only way DevOps is going to remain relevant and address the various challenges that can come before projects.

This natural alignment between the seven-step improvement process and DevOps comes in handy as the process can be readily, in its present form, applied to any aspect of a DevOps project and get tangible results out of the cycle. As the nature is cyclic and iterative, we can probably read into the process as a step toward the future and a stepping stone into the DevOps world.

CHAPTER 5

Teams and Structures

Processes provide the guidance around the objectives to be met and the outcomes to deliver. They provide all the directions necessary for an organization to succeed. But, processes are not executed by machines (mostly). People are required to carry out the tasks set forth. People are extremely unpredictable, and our analog nature does not guarantee outcomes no matter how well the processes have been laid out. Therefore, it is imperative that the teams be set up for maximum chances of success. The setup for success starts with the team structure as the team's motivation, orientation, and enthusiasm stem from the home base. The teams and their structure are the focus of this chapter, which is aimed at delivering faster and with higher accuracy.

A Plunge into ITIL Functions

Functions in ITIL are the silos that teams are spread across. Each of the functions represents a grouping of people with different skillsets. As introduced in Chapter 1, the functions in ITIL are represented in Figure 5-1.

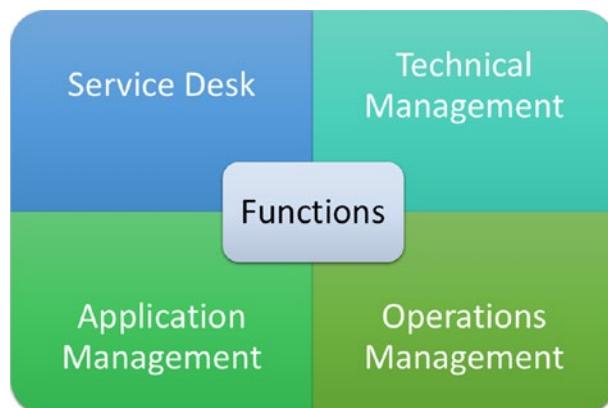


Figure 5-1. Functions in ITIL

Service Desk

The service desk is an integral component of the ITIL framework, and most ITIL implementations prioritize the design and implementation of the service desk and its associated processes. It is a group of people who act as the face of the service provider organization to users, suppliers, other service providers, and even customers. The service desk is the single and first point of contact for the identified stakeholders. If you have a problem with your mobile phone bill, you call your cell phone service provider, and the person on the other end of the line is from the service desk.

As the service desk serves as the single and first point of contact, it often becomes the face of the service provider organization. Therefore, it becomes an activity of utmost gravity to ensure that the service provider is fit to represent the service provider organization with professional etiquette. Generally, an organization's image depends on the service desk. Just think about it. If the service desk of your cell phone service provider gives you a cold shoulder for a genuine problem that you are facing, do you measure the service provider's performance based on this interaction? Definitely you would, even if the service had been immaculate up until now, because a single interaction can break a solid foundation built over the years.

The service desk serves as the first line of support for the service provider organization apart from being the first point of contact. If the service desk is unable to support the resolution of an incident or fulfilment of a service request, it gets escalated to the second line of support and then the third if the second line is not in a position to resolve. In an organization that supports an application, generally the first and second lines of support offer configuration changes. If the resolution requires changes to the code, it gets pushed to the third line of support. The third line of support is a part of the DevOps team in a DevOps organization. This is represented in Figure 5-2.

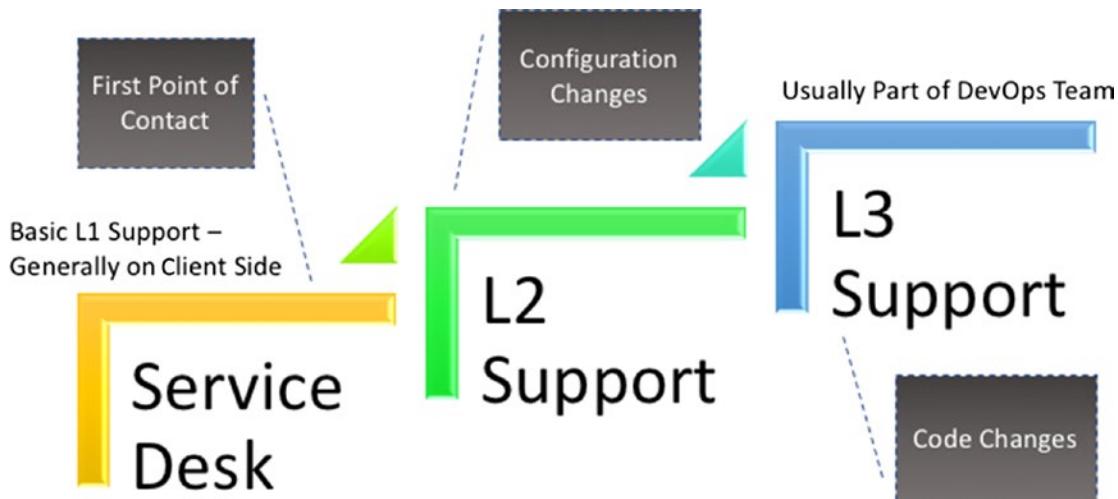


Figure 5-2. Functional escalation from service desk to L3 support

I will discuss the DevOps team and its role in the overall service desk throughout the rest of the chapter.

Note There are multiple types of service desks that are in vogue. The most prominent ones are local service desks that serve a particular location, centralized service desks that are located centrally to serve multiple locations, and distributed (virtual) service desks that are spread across multiple locations and, depending on the request type of requestor's geography, the call gets routed accordingly.

Technical Management

Technical management is a functional grouping of all the technical (infrastructure) teams. The organization structure is built around the specialization of individuals. The thinking behind this organization structure is to place people with similar skillsets in groups. Server teams, network teams, database teams, and data center teams, among others, usually make up technical management function. In Figure 5-2, the L2 and L3 teams do possibly come from the technical management function if the incident is infrastructure-related.

CHAPTER 5 TEAMS AND STRUCTURES

In Figure 5-3, the word *cloud* throws light on the possible technical management teams.



Figure 5-3. Technical management teams

The premise behind setting up a separate technical management silo is to ensure that the functional grouping will promote knowledge sharing between professionals with similar skillsets. This silo is their home, the base where they are groomed, mentored, and matured over time. When a project or a process activity comes calling, they are deployed into action. After completion of the project or process activity, they return to their familiar habitat.

This type of an organization is called a *matrix organization* where the people are placed in silos and are deployed into projects as and when needed. When the project is done, they go back to their silo. They typically have dual reporting, one from the silo and the other from the project. This is not ideal but is most commonly employed in most organizations.

The ITIL framework promotes placing people with similar skillsets in separate teams or silos. They don't make a distinction between people who have stepped into the field and the architects. Everybody is in the same silo. The rational is that the architects require the help of the operational professionals to design better, and the maintenance personnel may require the support of architects in their activities. However, most organizations today do not subscribe to this organizational framework but rather keep architects and operations separate.

Application Management

Technical management is to IT infrastructure is application management is to applications. Similar to technical management catering to IT infrastructure, application management specializes in software management (development and support).

Technical Management:IT Infrastructure::Application Management:Applications

Application management is a function that houses professionals who work on software development and maintenance activities. They are a silo that provides resources for various software-related activities involving application practices such as requirement gathering, analysis, design, coding, and testing. Figure 5-4 illustrates a few application management teams that could exist.



Figure 5-4. Application management teams

Apart from providing resources to meet process objectives, one of the main tasks for application management is to make decisions on buying third-party software or building it in-house. Based on the requirements, options available on the market, commercials, and other factors, the application management function is expected to make a decision whether the service provider is better served buying commercial off-the-shelf (COTS) software or building it. For example, if an organization needs a ticket management system and looks at the COTS options, ServiceNow, BMC Remedy, or HP Service Manager might be a better bet than building a product with in-house developers. But if an organization needs a specific workflow management solution and if the customization on the available COTS products is expensive and time-consuming, then it makes full sense to opt for an in-house solution.

Within application management, there are two areas: application development and application management. Application development is involved in developing new software or enhancing existing software, whereas application management maintains the software in its designed state. They take care of incidents as they come and don't get into the specifics of making significant changes to the software. The distinction in ITIL is very clear; you need people with different attitudes and skillsets to carry out application development and application management activities. It is prudent to identify the resources and move them into their respective silos.

The thinking behind different silos for application development and application management is in stark contrast with what DevOps believes in, which is "one team, one software." I will talk about a DevOps team and its objectives later in this chapter.

IT Operations Management

Operations is often seen as a launch pad for professionals getting into the IT sector. It is viewed from the prism of complexity and is rated pretty low, which is the primary reason for professionals to be put into operations, which signifies the bottom rung of the IT career ladder. However, this notion is not true. Operations cannot be done by everybody, especially newbies. It is a specialist job and requires a different set of skills to expect the unexpected and to learn rapidly from mistakes. For operational roles, you need people who are wired different so they can understand the underlying root causes from repetitive actions and the analysis carried forth to look for permanent solutions. In ITIL, the IT operations management function is tasked with managing the service operations.

There is a lot of strategy in the beginning followed by planning and implementation. But the actual execution of the plans are done by the IT operations management function. People under operations management interact fairly regularly with customers, and their attitude and zeal to keep the engine running can have an effect on the customer.

Within operations, the ITIL framework defines two distinct subfunctions.

- IT operations control
- Facilities management

IT Operations Control

IT operations control is the subfunction where the execution of operations for the IT infrastructure takes place. I mention specifically IT infrastructure because the application management group handles the operations of software-related operational activities.

Some frequent activities that are performed here are monitoring, running operational bridges, taking backups and performing restorations, managing batch jobs, printing, and keeping an eye on the performance of all systems.

A number of operations activities today can be fully automated to ensure efficiency and standardization and to avoid human errors and biases. Automation is one of the key aspects of DevOps where we try to avoid people carrying out repetitive activities with the sole aim of utilizing people power where it is really needed.

Operations in an integral part of DevOps teams, and the scope of the IT operations control function is purely based on the operational activities pertaining to IT infrastructure. Within a DevOps team, operations folk find a place, but they are mostly from the application maintenance front. Managing physical IT infrastructure is done outside a DevOps team as it is a common activity that serves multiple DevOps teams. However, within the DevOps teams, the IT infrastructure operations teams is involved in spinning up environments, managing them, and ensuring that the environments (or lack of) don't end up being a bottleneck for speedy delivery. This is all made possible through advancement in technology where servers can be created through code—a concept known as infrastructure as code (IaC). Environments can be created by writing scripts, and with the help of tools such as Puppet and Ansible, they can be spun up within a matter of minutes and hours as opposed to weeks and months of efforts for gaining approvals and manually processing environments.

Facilities Management

The second subfunction within IT operations management is facilities management. As the name suggests, this function takes care of facilities that are key to running IT services, such as data centers, control rooms, workspaces, recovery sites, and generators, among others.

The facilities management function is an overarching entity for all the teams and people involved in the delivery of IT services. They are the enablers for DevOps and other teams to deliver. They remain independent and outside the purview of DevOps teams.

DevOps Team Structure Revisited

In Chapter 1, I briefly introduced a DevOps team, as illustrated in Figure 5-5.

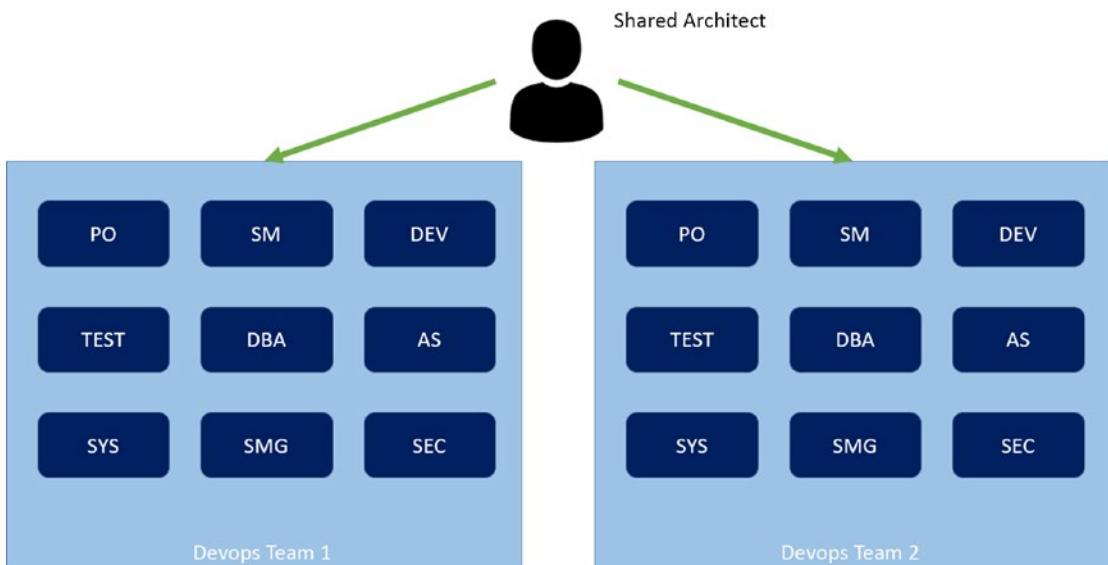


Figure 5-5. DevOps team structure

On the DevOps team, everybody associated with the product development is put together on the same team and is asked to make the best use of the aligned goals and objectives to intensify delivery and minimize rework (generally owing to defects). This structure is inspired by the Toyota Production System concept of Obeya (from the Japanese 大部屋, which translates to “large room” or “war room”) where in the time of crisis, all stakeholders are brought into the same room to quicken decision-making. When all the decision-makers are sitting across each other, there arises no need to wait for decision-makers to give their okays at their leisure. Likewise, when all the people connected to the development and support of a project are put together in the same room, the need to formally hand decisions between teams to “pass the buck” does not

arise. Given that the shared responsibility is enforced, people within a DevOps team cannot point fingers at each other as everybody becomes responsible for everything delivered, or a lack thereof.

Traditional Model

Let's examine how this is different from the traditional sense of setting up teams that is defined in the ITIL functions and in waterfall's matrix organization. In a traditional organization, people with similar skillsets are placed within the same teams and verticals and are called into action as and when needed. The people are always temporarily assigned to projects and are pulled back to the home base at the end of the engagement.

Consider Figure 5-6, which depicts a typical matrix organization where practices denote various verticals consisting of resources with similar skillsets. In the illustration, I have considered development, testing, server, database, project management, and architecture the practices. Digging deeper, the development practice will have multiple teams based on the technology that I have generically called team 1, team 2, and team 3. This could be a Java team, Microsoft team, web development team, and so on. Likewise, each of the practices can be subdivided based on technology, which is the most common practice of organizing teams within enterprises.



Figure 5-6. *Matrix organization*

Let's say that a project team is mobilized to develop an application. Some of the team members from each of the practices come together to form a project team. This is illustrated in Figure 5-7. In this illustration, I have picked up developers, testers, database

CHAPTER 5 TEAMS AND STRUCTURES

admins, an architect, and a project manager. Note that the server administrators from the server practice were not put onto the project team because the infrastructure is considered as an enabler and it stays outside the project governance. During project planning, the infrastructure setup is considered a dependency, and this dependency is likely to turn into a risk in due time since a discrete governance model is a formula for failure. Likewise, there could be other dependencies, but considering that infrastructure is most essential, keeping it outside from the project governance is a risk waiting to materialize.

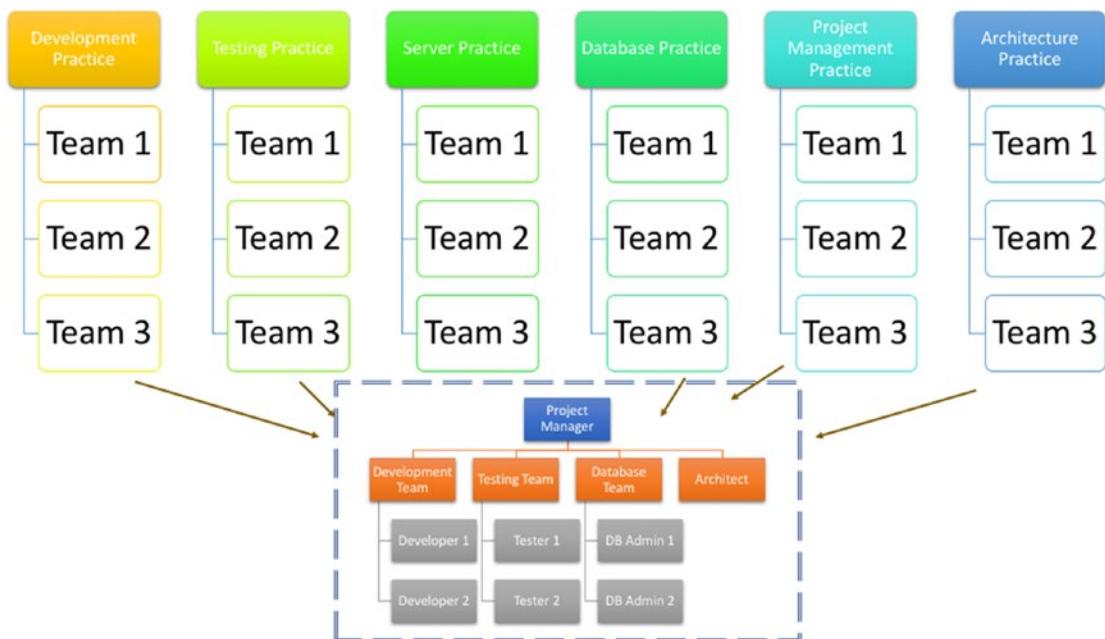


Figure 5-7. Mobilizing a project team

People from different practices come together to form a project team and then you can notice that a mini hierarchy of sorts has been set up within the project team. There is a development team consisting of coders, headed by a development lead. Likewise, testing leads and database leads head their respective teams. The structure is hierarchical because of the influence from the organization's hierarchical setup. Waterfall project managers also argue that this is an ideal structure to avoid a conflict of interest. The testing team, for example, will be completely impartial when testing the functionalities as they are under different leadership, and they are not obliged to push it through. I will examine this argument under the Agile model.

Agile Model

The Agile model, more specifically the Scrum framework, guides us to maintain smaller teams, and each of these teams is homogenous and hybrid with developers, testers, a Scrum master, and a product owner. These teams are called Scrum teams in the Scrum framework. This is illustrated in Figure 5-8.

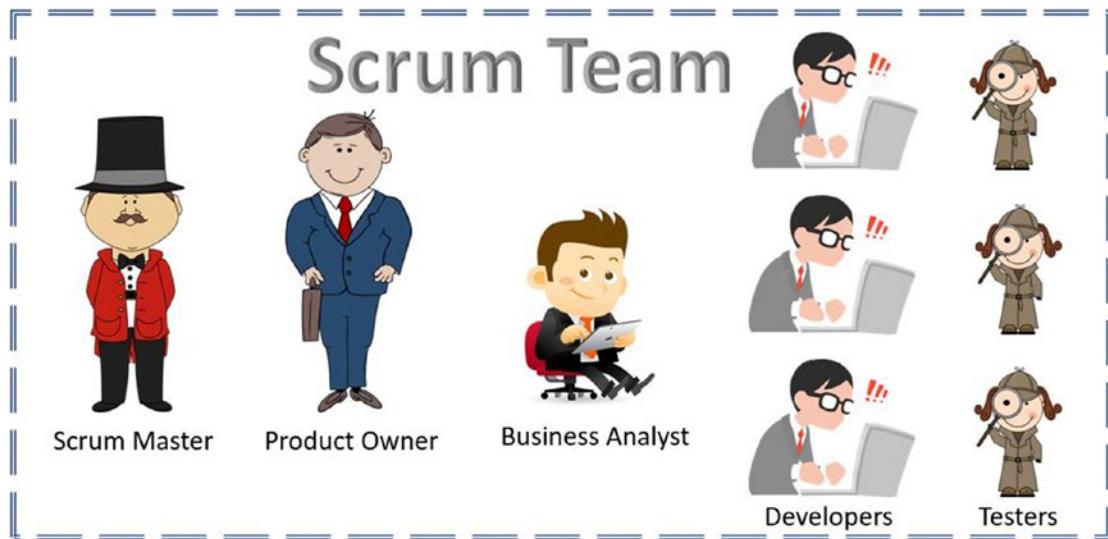


Figure 5-8. Scrum team

There are a few seemingly major differences between a traditional model of team organization and a Scrum-based organization.

Flat Hierarchy

The Agile team organization is as flat as a pancake. There are no layers as clearly visible in the traditional model. Flat organizations help transmit and receive information between team members because of the lack of hierarchy, which helps promote better collaboration between team members. Collaboration is one of the key asks of the Agile framework where the entire team works as a single unit.

A flat organization is not the panacea for all of an organization's hierarchical problems. It works the best if the teams are smaller and are well-adapted to freely sharing information. Therefore, in this flat structure, in the Scrum team, the recommended team size is between six to nine members—that's it! These numbers are based on experiments and the experiences of various project management experts in the field.

How will you manage with a maximum of nine resources for bigger projects? you might ask. The answer is to form multiple Scrum teams with homogenized setups. In other words, every Scrum team will have a dedicated Scrum master, a product owner, a business analyst, developers, and testers. Each Scrum team will be responsible for the development of a particular feature or a module to ensure that they work independently with minimum dependencies as much as possible. For example, if you are building an Internet banking application, Scrum team 1 is tasked with the development of fund transfers, Scrum team 2 with credit rating checks, and Scrum team 3 with the debits and credits feature. All three are going to come together to form a single application, but managing each of these modules or features independently within a homogenized team provides the project team with the best chance of success.

No Project Manager

In Figure 5-8 you might have noticed that there is no project manager. This is not a mistake but rather by design. You don't need project managers if the team is self-supervised and is able to work synchronously on their own. To help with the synchrony, a new role of a Scrum master has been introduced. The Scrum master, unlike the project manager, is not a manager who keeps people in check while they work. The primary role of a Scrum master is to help the Scrum team in their developmental activities by removing impediments that come in the way of hurdles. A Scrum master is like a servant leader who leads the team members by helping them to succeed. Think about it as a cross-country relay where the racers are required to cross several bumpy terrains to win the race. The Scrum master essentially removes the bumps from the way of the racers to ensure that they drive swiftly and come out on top. The actions are generally attributed to leaders, and the Scrum master is doing it not by commands but rather by service.

Single Team

In the traditional model there were mini hierarchies: development team, testing team, and database team. The logic is to keep each of these teams independent of the others to ensure ample wriggle room to do their work and to ensure there is no conflict of interest. The traditional modelers did not want to have a situation where the testers would pass all the tests owing to the pressure of release timelines. The logic is sound, but can there be a better way to manage it? On the downside, handovers between each of these teams were sluggish and often not seamless. A lot of time and effort went to waste

in trying to explain the reasoning behind the codes and the test failures between teams. For example, during the coding phase, only the coders are given a low-down of the requirements, and then the coders pass on the developed code to the testers and at that stage explain what the requirements are. How much of the information is passed in this game of Chinese whispers? The testers tested based on what they understood but from second and third parties.

The answer to this problem that materialized more often than not was to build a single team with all the roles fused in. We create a single team and not multiple teams as was the case in the traditional team. Remember that the teams are small in the Agile case, so a large traditional model team will typically translate into multiple Scrum teams. When you fuse team members with different skillsets into a single team, the next thing on the agenda is to provide shared responsibility, where everybody in the team has equal ownership in the delivery of the product or the feature. This means you cannot have a situation where the developers did extremely well but the testers failed to test it on time. Where there are any kind of lapses at any stage of the development lifecycle, the entire Scrum team takes the responsibility for nondelivery. When shared responsibility is made the principle for judging the team members on their work products, conflicts of interest simply disappear. They do not arise because the entire team wants to see the product succeed and will not be in a position to compromise what they see as substandard.

Product Owner

Apart from the role of a Scrum master, which is new, there is yet another role introduced in the Agile organization: the product owner. The product owner is the sole owner of the product backlog, which is the list of requirements that need to be developed. The product backlog consists of all the nuts and bolts that need to go into the product or the feature. The product owner owns this list and is the person who prioritizes this list to help the Scrum team pick up items from the product backlog during sprints. If any clarifications are needed on the requirements, the product owner is ready to help the Scrum team.

How is it that the product owner has a complete grasp of the requirements, and how is that the product owner knows what priority is to be picked up? you might ask. The product owner is part of the Scrum team but generally is from the business organization or from the customer organization. The proximity of a business representative who owns the requirements list helps the development team develop better, faster, and to the requirements. In the traditional model, the customer always sat outside the

project organization and would be brought in only during testing and demonstrations, which would be at the end of the project lifecycle. In the Agile model, the customer or a customer's representative takes a seat on the project team and works with the development team closely in realizing the requirements. This change in stance in terms of where the customer sits will avoid the gulf of mismatch between what was asked and what was delivered.

Predictability

Predictability may not be a good feature for individuals, but for projects, it is a top-notch character trait to possess. In a traditional model, a project might run for a few weeks, and another project might drag on for a year or two. The cycles of development, testing, and other associated activities happen at different times depending on the project plan which is carefully laid out well before the project is due to begin. Over a period of time, contexts change, and the project plan changes along with it. This seems to bring in plenty of uncertainty to the mix with long project timelines and multi-angle changes bearing their ugly effect on the deliveries.

The top motto of the Agile project management methodology is not to freeze everything at the beginning but to keep things fluid, transparent, and flexible to adapt to things that change. Predictability is the last thing you would associate with Agile, right? Wrong! Although Agile is flexible to pick up whatever gets pushed up the list by the product owner, most elements in the framework have a predictable nature. For example, the sprint length is two weeks, starting on the first Monday and ending on the second Friday. On the first day of the sprint cycle, we have a sprint planning session where the entire team come together to estimate and plan what can be achieved during the two weeks. On the last day of the sprint (second Friday), the developed product is showcased to stakeholders, and feedback is received. Also, on the same day, the team sits together and does a post-mortem called a *sprint retrospective* to understand what went right and what went wrong during the sprint and to identify ways of not repeating the mistakes and to optimize the delivery. So, come what may, every second Friday, the customer stakeholders are aware that a demonstration of the developed product (however small) is on schedule. No matter what happens in the market, the sprint length remains the same, and the sprint plans are drawn up for the two-week estimates and delivered accordingly within the two-week window. This is the predictable side of Agile project management, which provides a road map for customers on what to expect and what to commit.

DevOps Model

The Agile model is self-sufficient. It is a single homogenous team that has all the elements of development built into it. Once the software is released into production and gets accepted by operations, the Agile team kind of gives up its baby for adoption and washes its hands of it. The team that developed it knows the product intimately, so why give it up to another team to manage? Logic does not support it, but there are certain sections of the IT community that believe operations is rookie work and experienced hands shouldn't be doing it.

Let's take this argument one step further. If this product were to break down, experienced hands require X amount of time to recover it, and the rookies because of their relative inexperience of the product require about 4X time (the numbers put forth in this argument are factionary; there are no studies to suggest that it takes four times the effort). The amount of time taken to resolve the issue is the downtime during which customers are unable to use the product or the service. This downtime can translate into penalties based on the SLAs, negative perception of the service provider, and bad-mouthing of the service provider to other potential customers. Given that the time taken to resolve an issue can take significantly a lot more time with a separate operations team, will service provider organizations sacrifice perception and avoid penalties for the sake of engaging experienced hands only on developmental activities? This sounds absurd! This is one of the prime reasons why the concept of DevOps took shape and the DevOps team was born to break all barriers that exist.

Composition of a DevOps Team

The objective of a DevOps team is to build a team that is the alpha and the omega for a product or a service. We look at this team for all its needs. The thinking behind this is not to scatter the knowledge across teams but to groom and built it under a single umbrella that is close quartered.

The DevOps team that I introduced in Chapter 1 is the target team we achieve to build that has both the development and operational team members. It is a team that is going to co-exist between sections of IT that are considered to be on different poles, the development and operations teams.

Another illustration of a DevOps team is provided in Figure 5-9. The DevOps team is a conglomerate of three different teams. The first is the Scrum team that we discussed in the previous section under the Agile model. The second is the application

management function from ITIL; this is the team that is involved in managing and maintaining the application. The final parts of the DevOps team are the other functions that help in nonfunctional aspects of a product, such as IT security, tools automation and configuration, and service managers who manage the various service management activities and also act as a conduit between the DevOps team and the customer organization on operational aspects. The formation of a DevOps team is decided by the product, the service, the service provider organization, and the contracts that were signed. The team composition presented here is for illustration purposes only. For example, if you don't need a service manager to be part of the DevOps, so be it. Somebody else can pitch in to take up the role, or maybe a shared team can manage service management activities. The same is true for all activities coming under other functions. However, the roles coming under the Scrum team and the application management teams are a fairly standard setup for a DevOps team, and it can be said that they are the permanent members of the team.

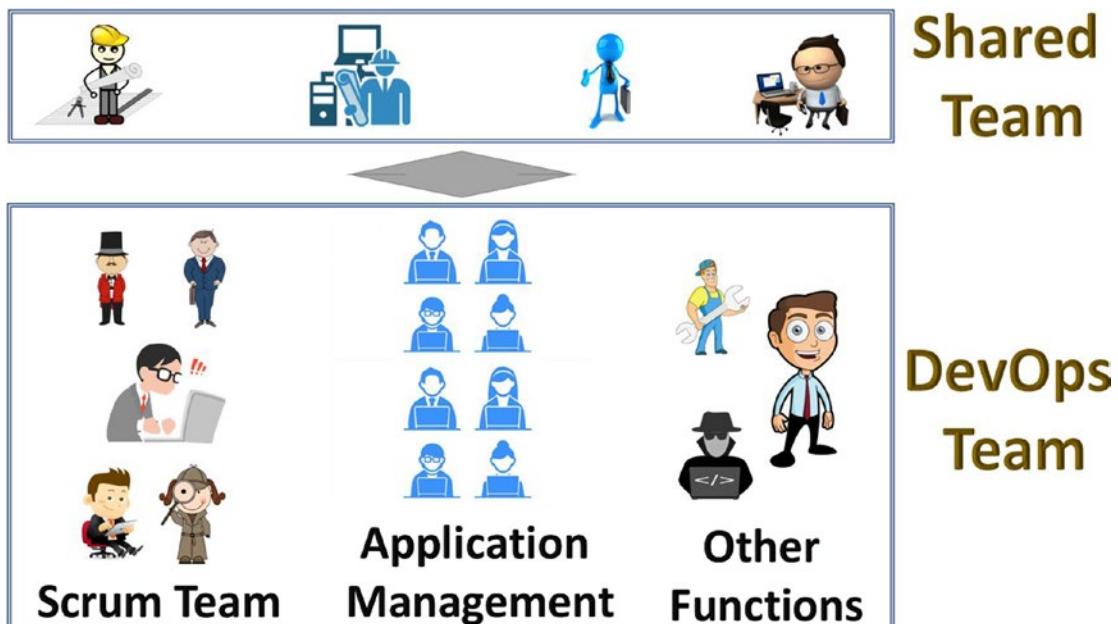


Figure 5-9. DevOps team composition

There is another team that I have introduced in Figure 5-9, the shared team. The shared team consists of a set of functions that are necessary to be carried out in a DevOps workload, but a dedicated team member carrying out the function is not the best use of that person's costs. Therefore, we create a shared team for functions such as architecture, IT infrastructure, consultants, and domain experts where they support multiple DevOps teams. In effect, their time will be shared between multiple DevOps teams, and they are required to manage their time and expectations with the DevOps teams on their availability and delivery. I have always found this challenging, especially when you have significant deliveries to make. But we don't have a choice today with the onus on cost reductions; for example, a dedicated consultant or a dedicated domain expert is a major waste of resources if they don't have full-time work.

Consider Figure 5-10 featuring the scope of a team's reach within an organization to provide a sense of how people are spread across teams.



Figure 5-10. Teams' scope in an organization

The DevOps teams are on the lowest rung in terms of the breadth of scope covered, meaning that they are pretty much restricted to the product and service they develop and manage. They are focused on their product, and they are 100 percent dedicated to supporting the product placed in their scope. They don't get into the development or problem-solving of other products and services.

The shared teams, on the other hand, have a wider reach than a DevOps team. The name *shared teams* suggests that the resources within the shared teams support multiple DevOps teams. For example, a software architect may provide architectural support to multiple DevOps teams, which is fairly common.

The next layer in the hierarchy of scopes consists of the umbrella teams. These are teams that carry out their work across multiple projects and services. They don't restrict themselves to limited teams but rather do it for an entire program or an organization depending on the size of the organization. Some examples include the capacity management activity, which is generally done at a larger level considering that the capacities of infrastructure and network are generally shared across multiple projects. Another example is that of the asset management activity. Managing assets (like laptops) is generally governed from an organizational level or a business unit level.

The topmost rung with the maximum scope space consists of strategy and compliance. All the strategic and compliance activities are carried out across the organization by default rather than at a lower level. Organization's strategy will affect all projects, DevOps or not, so it is at the highest rung on the scope-o-meter. Compliance as well is strategized and planned at a higher level, although it will come down to individual projects being audited and put under the lens.

ITIL Role Mapping in a DevOps World

ITIL brings in a number of roles across its five publications. Figure 5-11 shows the list of roles from the service provider organization. I am deliberately not listing the practitioner roles and other roles that are from the customer and user community as they have no bearing on the mapping with the DevOps methodology. Also, the lower-rung roles such as coordination roles usually exist in organizations but not in the publications. Also, you will not find the list of roles that are coming from functions as I have discussed them in detail in the earlier sections.

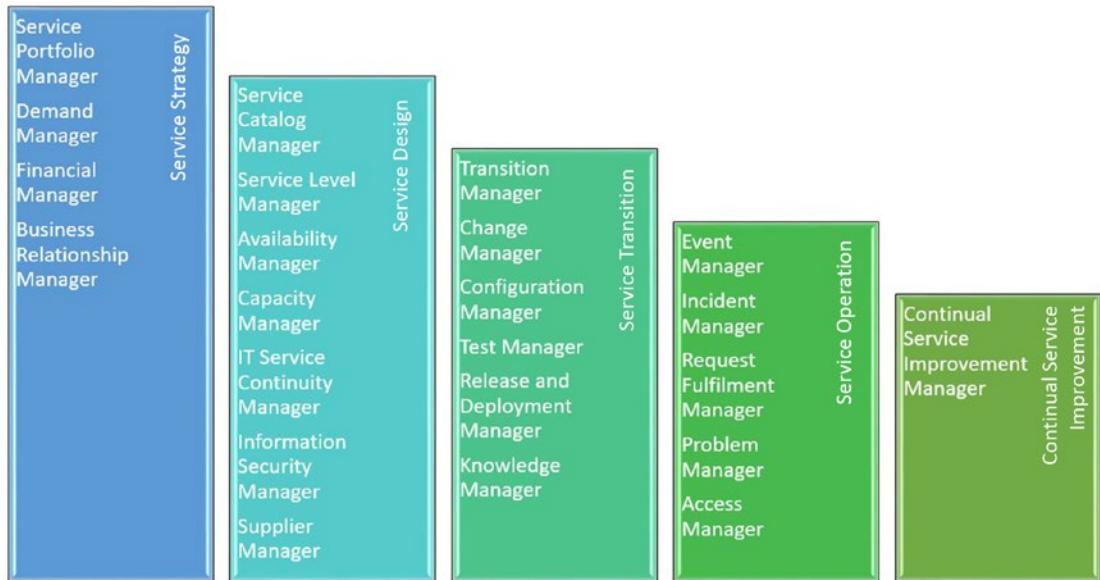


Figure 5-11. ITIL roles

Strategy and Compliance

The roles that appear in the service strategy lifecycle phase of ITIL can directly be mapped to the strategy and compliance role structure in the DevOps model as the ITIL roles performed are at a strategic level and the DevOps scoping is for strategic activities along with compliance.

Figure 5-12 indicates the ITIL service strategy roles in the DevOps scope of activities. The role of a quality manager is not explicitly named in the ITIL publications. However, to manage the compliance of related ISO standards such as ISO 20000 and ISO 27001, we need a head to lead the quality function.



Figure 5-12. Roles under strategy and compliance in the DevOps scope

Umbrella Teams

Umbrella teams are the set of activities that you perform that affects the entire organization or the business unit. The activities performed here are common across the board, and they level the playing field. For example, a change policy/process is generally common for all technologies and business units in mature organizations because this is the process that controls what goes in and what doesn't. I will talk more about it later in the book. Figure 5-13 shows the roles that are generally placed in umbrella teams.

Most of the ITIL roles go under the umbrella teams scope as the activities they manage are common across the entire organization or business, and it makes more sense to keep them standardized. The criteria for roles to be placed under the umbrella teams is that the policies and processes stay common across the organization or the business unit.

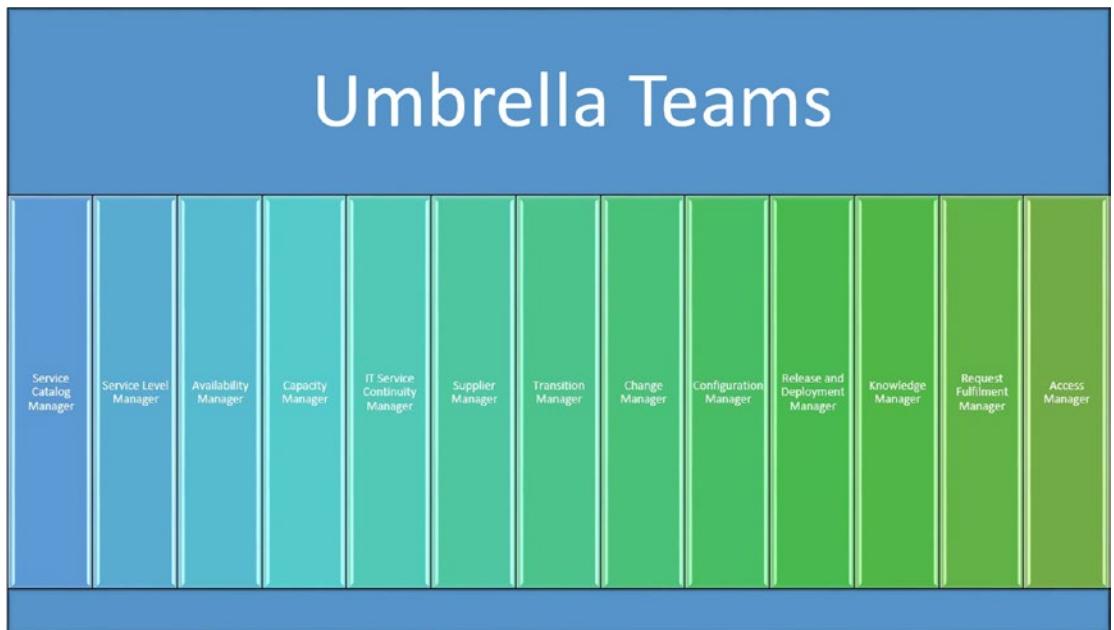


Figure 5-13. Roles under umbrella teams scope

Shared Teams

Shared teams work at a DevOps team level but across multiple DevOps teams. The strategy behind coming up with the shared teams is to ensure that costs are splurged on various roles, especially when the workload is not 100 percent. Some of the roles that I have indicated in Figure 5-14 can easily move into the DevOps team scope depending on the product and the scale of the project.

You will find that some of the roles such as configuration manager and knowledge manager find representation under shared teams as well; they previously featured under umbrella teams. They find double representation because the role under umbrella teams sets the tone for how the configurations and knowledge databases are developed and managed, and the execution is done at a shared teams scope.

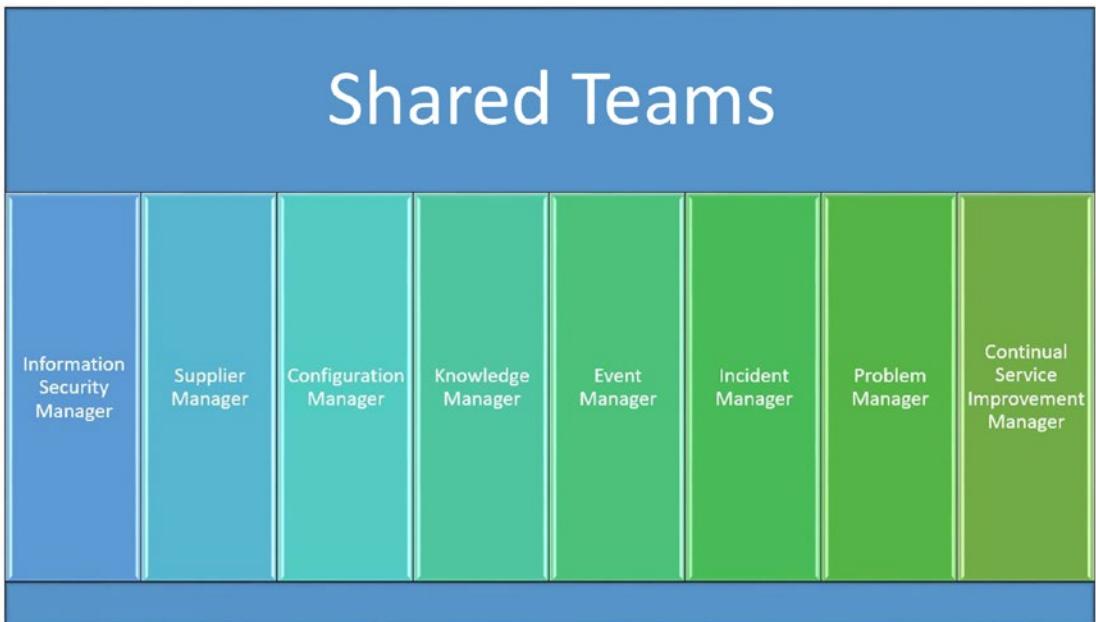


Figure 5-14. Roles under shared teams scope

DevOps Teams

The DevOps teams are mostly made up of people who work on the ground, close to the development and operations teams. You will find that some roles are common with the shared teams and umbrella teams. If a specific role is in the DevOps teams, then it can be absent from the shared teams scope, and vice versa. Figure 5-15 showcases the roles under the DevOps team's scope.

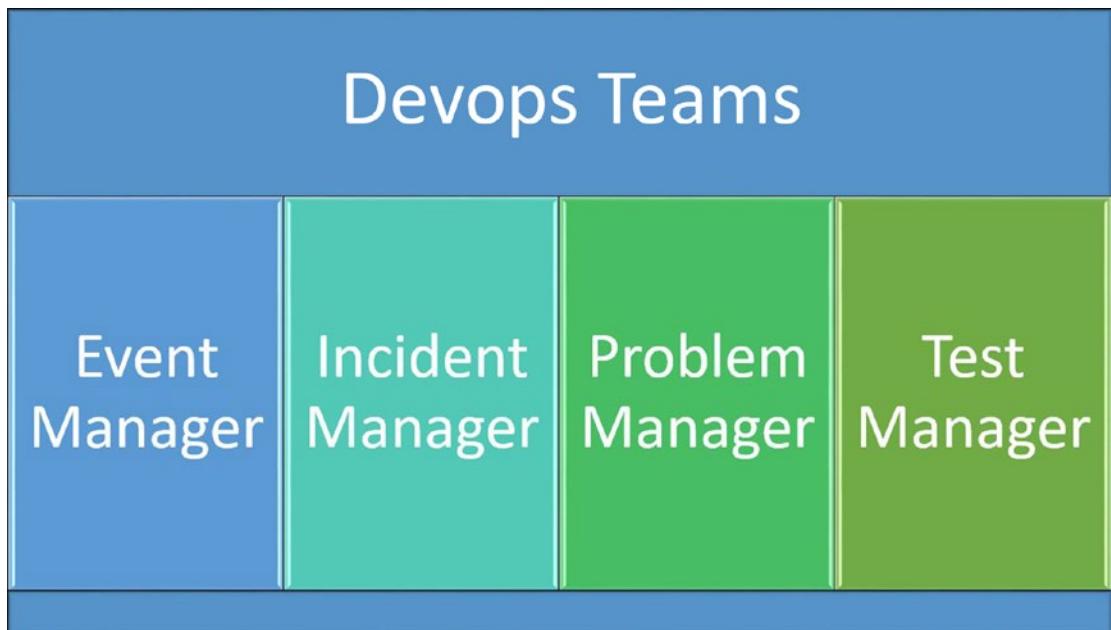


Figure 5-15. Roles under DevOps teams scope

The mapping that I have offered is to be taken as guidance because every organization has its own chemistry and trying to combine a standard set of elements goes against the nature of DevOps.

CHAPTER 6

Managing Configurations in a DevOps Project

Most projects fail because of the lack of managing configurations and having total control over the various components that make a project tick. Configuration management is the foundation upon which the entire project is built. Giving no care to the foundation will logically see the walls and roofs of the project crumbling down in record time. Configuration management plays a significant role for systems made up of multiple components that are integrated with other systems and run on multiple dependencies. Sound familiar? Yes, most systems today are complex owing to the need for integration and its respective data sources and data consumers. In such a complicated setup, it is imperative that systems are driven by configuration management, which is relied on heavily by projects, which, in our context, are DevOps projects.

ITIL's service asset and configuration management process has matured over the years and has been powering the service industry for a number of years. I will delve into the ITIL configuration management process in the first half of this chapter. I will follow it up with what constitutes configuration management in a DevOps project and provide the details around how they can work in unison.

ITIL Service Asset and Configuration Management Process

The ITIL service asset and configuration management process has served as the spine for IT services over the years, and within ITIL, the process has matured with each version. It is a process that defines whether a service provider succeeds or not in delivering IT services and defines the longitude of the services offered.

The concept of configuration management is simple enough. Configuration management gives you a blueprint of IT services, the architecture underneath, and the dependencies. It provides an accurate reflection of the connected pieces and dependencies. It is this network of components that make the service work, and having it in a form that's alive and accessible gives the ammunition to make changes to services with ease, identify business impact with minimal analysis, and resolve outages in a jiffy. These are the tools that you need to be a valid player in today's market where changes happen on the fly and customer wish lists change faster than the lifespan of mayflies.

A project without accurate and dynamic configuration management is a living nightmare. Imagine making changes to one part of the system without understanding the impact they could cause on other dependent systems. This happens commonly in the software development industry. It is not uncommon that architects are baffled when they have certain dependencies and defects crop up through the regression of acceptance testing quite late in the development lifecycle. This is a blunder of sorts because there is a good likelihood that software delivery might not happen as per the promised schedule, and if the development teams try to cram in fixes at the last minute, defects pop up. If only the architects had a working configuration management process in place, they could have identified everything that needed to be changed and could've avoided the negativity that emanates from failures.

Objectives and Principles

In short, the service asset and configuration management process exists to ensure that the various moving parts of a service are identified, registered, and maintained as long as the service is operational. The principle behind the process is to identify the smallest piece of a component that can be uniquely identified and managed and to connect such components with each other to build a service model. Once a service model is built, it needs to be maintained as long as needed to support the restoration of services, to identify the business impact, and to make changes to the service.

Service Assets and Configuration Items

There are two parts to this process: service assets and configuration items (CIs). Service assets are individual elements that make up a service. The entire lifecycle of the service assets—beginning with initiation, the changes it undergoes, and finally retirement—is managed, controlled, and tracked in the service asset and configuration management

process. Examples of service assets are software, licenses, monitors, laptops, datacenters, and servers.

A service asset is any resource or capability that could contribute to the delivery of a service. A configuration item is a service asset that needs to be managed in order to deliver an IT service.

A configuration item is a fundamental component of a service that can be configured, tracked, accounted for, and controlled. For example, in an e-mail server involving servers, routers, and MS Exchange applications, each server, router, switch, application, and firewall can be considered a CI. Why? Because these CIs can be tracked, controlled, accounted for, and audited.

Not every service asset is a CI, but every CI is a service asset.

Who decides what can or cannot be a CI? This is a decision made by the configuration architect based on the nature of the services, its interfacing to other processes such as incident and change management processes, and most importantly the cost. For example, a server can be considered a CI. Conversely, each of the components of a server such as the processor, memory, and hard drives can be considered as CIs, which necessarily alludes to a lot more effort (and cost) in coming up with the configuration management and maintaining it. Therefore, generally the decision is left to the architect to make a judgment call on what level a CI should be considered. The general practice is to measure the value that is derived by delving deeper into the services for deriving CIs.

Every CI has a number of attributes attached to it. Attributes are various details that get recorded against a CI, such as owner, location, date of commission, status, and configuration. All these attributes are controlled through change management. This is the layer of control that ensures that the configuration management remains accurate and nobody can make changes to it without the approval and consent of the change management governance.

Any service asset that is critical or that directly impacts a service is a configuration item. This definition gives rise to a number of types of CIs that could potentially be leveraged in a service provider organization. Human CIs (workforce management), document CIs (document management), business CIs (the business processes that connects business side of things), software CIs (business applications and in-house developed software), and hardware CIs (server, router) are some examples. An architect can choose to include only IT elements (such as software and hardware) or go shopping for human CIs through HR departments and document CIs through document management teams. It is entirely the architect's decision on how to manage the CIs appropriately.

Scope of Service Asset and Configuration Management

As the name of the process suggests, there are two main parts to the service asset and configuration management (SACM) process: asset management and configuration management.

The asset management part of the SACM process is where the accountability of all the service assets happens. Under this, the service assets are identified, accounted for, managed, and controlled. The type of assets that will be individually managed will be at the discretion of the service provider organization.

For example, a service provider might decide to include the monitor as part of a desktop so as to not manage the monitor individually but under the whole unit of a desktop computer.

The question to ask is how well a service provider is able to manage the service assets, without any compromises to the users, to the service provider personnel, and to the services. Based on this, the service design lifecycle phase deems whether certain service assets are within or outside the scope of individual management. Remember that every single asset has to be managed. Whether they are managed individually or as a group is at the service provider's discretion.

In most organizations, service assets have a financial value associated with them. The user group that enjoys the service will be charged for the assets leveraged. For example, if I am using a laptop, each year my business unit gets billed a certain amount of money for the laptop that I use. Of course, it is notional charging, where an actual exchange of money does not take place. But, it is a good practice to keep track of assets and their financial information across the organizational units.

On the other hand, the scope of configuration management is based on services that potentially impact business by the lack of it, or even in its degraded state. The scoping of which services to be included in the configuration management process is decided by the architect.

Introducing the CMDB, CMS, DML, and DS

To manage the service asset and configuration management process, we rely on a number of databases with varying relevance and significances.

Configuration Management Database

A *configuration management database* (CMDB) is a repository containing all the CIs including their relationships. For example, in a CMDB model, the dependency between the CIs can be defined through relationships such as “runs on” and “supported by.”

Within the CMDB, you can have multiple services, the individual CIs, and their relationships. Most modern ITSM tools, such as ServiceNow and BMC Atrium, offer placeholders to record the upstream and downstream impacts. If you pick up a service and want to use it visually to see how CIs connect to one another, you will see an array of connections between the CIs. Using this visual image, other processes such as incident management can troubleshoot incidents with ease, and processes such as change management can identify upstream and downstream impacts with a click of a button. Imagine if this were not in place; the whole activity involving analysis and troubleshooting would be tough.

In an organization, you could have multiple CMDBs depending on the requirements, business structure, and customer obligations. For example, you can have a CMDB for business units A, B, and C; a CMDB separately for customer ABC; and yet another CMDB for internal infrastructure and software. There is no limit, as long as the logic makes sense to manage, control, and simplify matters.

Configuration Management System

The *configuration management system* (CMS) is the super database that contains all the CMDBs and more in its ecosystem. It is the layer that integrates all the individual CMDBs along with other databases in the IT service management space, such as known error databases, incident records, problem records, service request records, change records, and release records. Figure 6-1 provides an illustration of a CMS.

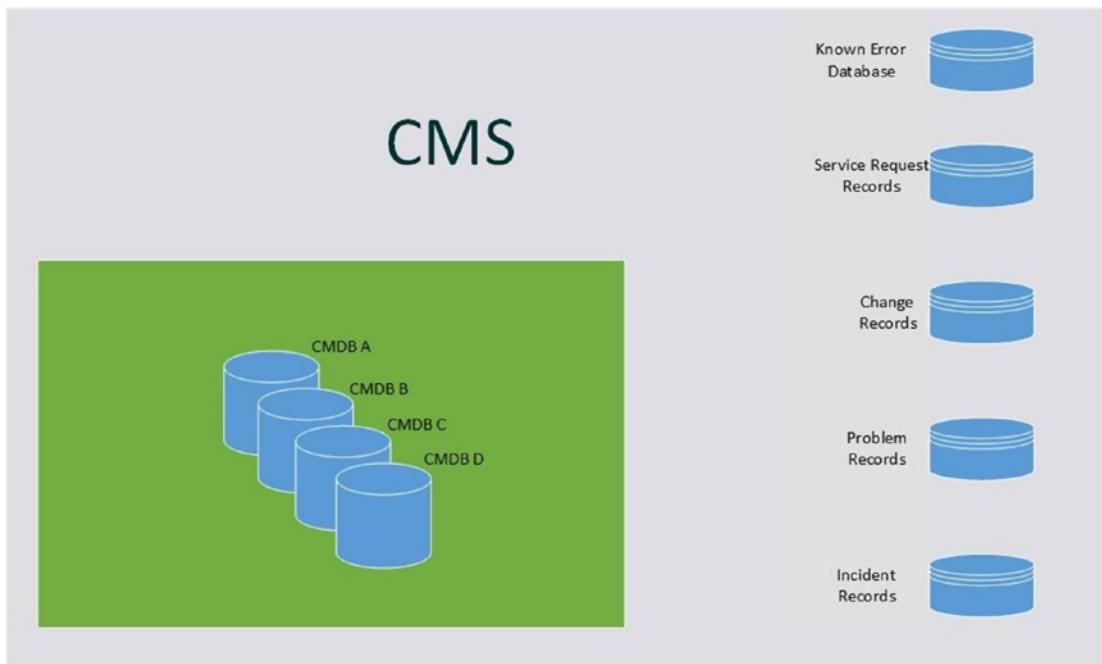


Figure 6-1. Configuration management system

It is possible that some CIs within a CMDB talk to other CIs in another CMDB. The overview of all the relationships between the CIs is provided in the CMS.

The CMS holds the CI data as well as other databases such as incident records, so service providers and customers alike can utilize the CMS in identifying all the incidents raised against a CI and the number of times they have failed on a regular basis.

Definitive Media Library and Definitive Spares

The *definitive media library* (DML) is a repository for storing all licensed copies of procured software and software that has been developed in-house. The repository can be an online one or a physical one, but it needs to be access controlled.

The software that gets accepted into the DML is controlled by the change management process, and only those copies that are authorized by change management into the DML are allowed to be used during the release and deployment management process. The software that gets into the DML is expected to be analyzed, quality tested, and checked for vulnerabilities before getting accepted.

In the case of a physical DML where CDs, DVDs, and other storage media are used, it is expected that the storage facility is fireproof, can withstand the normal rigors of nature, and is secure against media thefts.

The DML is ideally designed during the service design lifecycle phase, and the following are considered during the planning stages:

- Medium to be used and the location for the master copies to be stored
- Security arrangements for both online and offline storage
- Access rights, including who has access and how it is controlled
- The naming convention for the stored media to help in easy retrieval and tracking
- What types of software go into the DML, for example, source codes or packages
- Retention period
- Audit plan, checklist, and process
- Service continuity of DML if disaster strikes

The *definitive spares* (DS) is a repository for storing hardware spares. Generally, all organizations store a certain amount of stock, mostly pieces of critical infrastructure, to be used to quickly replace hardware in the case of an incident. Also, there are stocks that are needed for the operational consumption and ever-increasing demands of the customer.

Like DML, a DS must be secured, tracked, managed, and controlled. However, change management generally does not get involved in controlling the items that go in and out of the DS, as the gravity of compromising intellectual property and master copies of licensed versions can be messy compared to hardware spares. The SACM process oversees the overall functioning of the DS.

Service Asset and Configuration Management Processes

The service asset and configuration management process is a complex process with technical and logical flavors playing a big part in its definition and implementation. At a high level, the process can be explained in five steps, as shown in Figure 6-2. Translating these five steps into a full-fledged process is a different matter altogether. The architect behind configuration management must be well versed with the technicalities surrounding the service, have a good understanding of the contractual requirements, and must be an expert in ITIL to understand the interfaces of configuration management with other processes.

I have been developing configuration models and processes for a long time now, and still the field amazes me as there is always something new to learn, such as new technologies or a complicated contractual requirement such as a report on the number of cores for servers. I might write a book in the future on ITIL configuration management as the bookshelves look barren when it comes to configuration management and the techniques to implement it. In this section, I am only going to talk about it from a high-level view.

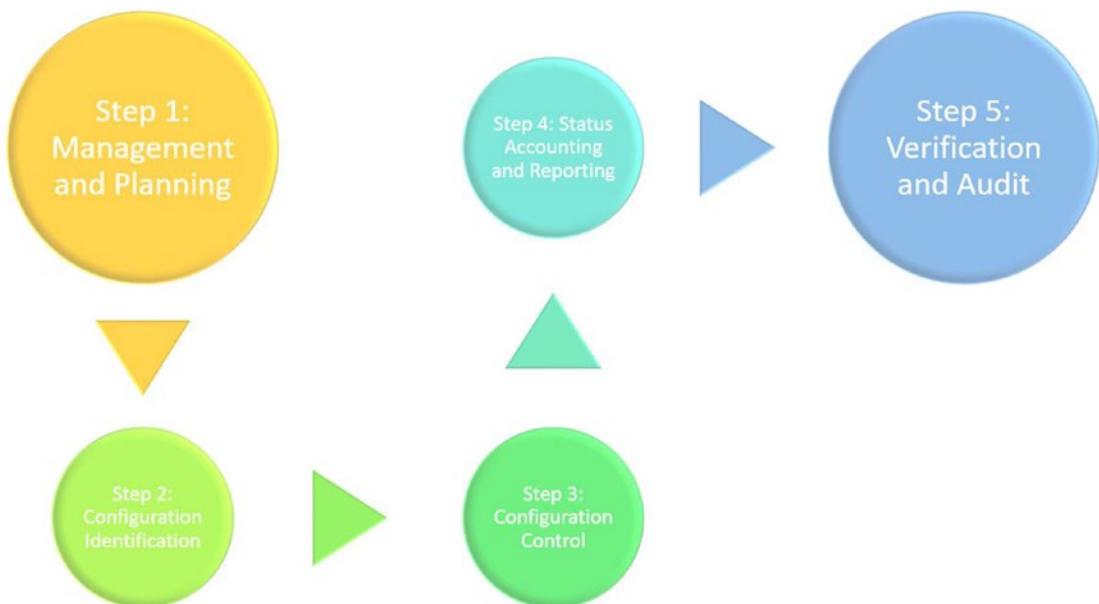


Figure 6-2. SACM process

Step 1: Management and Planning

Configuration plans play a major role in the SACM process. The plans lay out the various elements of the configuration management that needs considering in the implementation. There is no set template for plans to be formed, and every single plan is customized for the services in scope and the customer/interfacing process requirements.

A configuration architect will lead the planning exercise by giving a structure to the configuration model and defining various elements of configuration management that are fit for use and fit for purpose throughout the service management lifecycle. This is by no means an easy exercise. In terms of the timeline, a good plan might take as much time as the overall implementation of the process.

Before this step, the architect must identify all stakeholders and sit down with each one of them to understand the stated and unstated requirements. Contract documents are one of the major sources of requirements. After gathering all requirements and analyzing them for configuration management elements, a plan has to be defined with generally the following items:

- The scope of SACM (what services are in and which ones aren't)
- Sources of requirements and how they translate into specific configuration management decisions
- Interfaces with other process (specifying the exact nature along with the handshake details)
- Principles around which the configuration management will be built
- Identification of risks and dependencies
- Involved suppliers and their scope in the SACM process
- Identified tools
- Tools to be integrated between systems and suppliers
- Service assets and configuration items along with their attributes
- Configuration identification and control mechanisms
- Configuration management roles and their authorized accesses
- Other elements as needed

Step 2: Configuration Identification

A good plan gives way to a seamless identification of configuration items. Based on the identified configuration items and their respective attributes and data sources, the CIs are identified for the defined scope.

Identifying CIs is a tedious and time-consuming process, depending on the complexity of the services and the configuration management database. It is an activity where different parts of the organization must come together to identify the right set of CIs and register them accurately in the CMDB. For example, the data center teams, the tools team, and the configuration management team must work in close collaboration to capture the identified CIs from the data center into the CMDB. Tools help to capture CIs accurately and quickly. There was a time when we were capturing CIs based on the service architecture manually. Many times, by the time we acquired the architecture diagrams, captured the CIs, and registered them, the CIs involved would have been modified or replaced. Or new CIs could have been introduced into the architecture. This was a risk that we were running against in the manual identification of CIs.

However, today things are a lot more advanced, and most of the CIs are captured automatically using discovery toolsets such as Service Watch, ADDM, and Dynatrace. The discovery tools can even identify the relationships between CIs (such as a software using a database). The manual activity in this whole process is to validate the data to ensure its integrity.

Step 3: Configuration Control

When all the configuration items are identified and the service models are built, they are not going to remain constant forever. They can change at every turn, such as when tweaking configurations, replacing modules, or even changing the architecture. For the configuration management to stay relevant and useful, the CIs that are identified and built must remain accurate at all times. A tight control net must be weaved around the CMDB to ensure that all changes to the database happen through a defined process that is streamlined and without loopholes.

Since configuration identification takes a good amount of time, it is likely that the configurations will change before it officially gets into the configuration control stages. Therefore, it is practical to set up the control processes, and as soon as configurations are identified, they automatically go into the pipeline that's controlled by the defined

control processes. If control processes kick in sequentially, then it is likely that we will look down the barrel to find the changes in the CMDB even before we claim a stabilized CMDB build.

The control processes are generally concerned with the following:

- When the CMDB will be modified (trigger and input)
- How it will be done
- Who does it
- Handshakes with other processes

Generally speaking, a good configuration management control process will initiate making changes to the CMDB on the back of a change ticket. After making the changes to the CMDB, the data is then verified with the live environment to ensure that the change performed is as per the proposed change plan. For example, if a change ticket is raised to upgrade the hard drive on a server to 2TB, but instead the hard drive inserted is 1GB, configuration management ought to identify the mismatch and have mechanisms to flag it to all concerned. So, configuration management is not just a database to store data but is also a validation tool that can help organizations achieve total control.

Step 4: Status Accounting and Reporting

The configuration items that make up the CMDB have lifecycle states. Let's say that a server gets ordered, procured, and delivered to the service provider. When the server gets delivered, generally it is registered in the CMDB with the status of "in store." When it is getting built or tested, the status may change accordingly. When the server makes it to production, it takes the state "active" or "live." When it's time to send it for maintenance, the status gets changed accordingly. When its end-of-life dooms upon it, then the server gets decommissioned with a status of "decommissioned." Finally, it is discarded as e-waste, and the server ends up with a final status of "disposed."

All the change of statuses must be defined and controlled. For the status to change from decommissioned to disposed, let's say the acceptance criteria is that the hard drive is wiped clean and is degaussed. Unless this is done, the status will not change to disposed. Likewise, between the change of statuses, there are input, acceptance, trigger, and output criteria. The status accounting activity is responsible for ensuring that every change in state for a CI is recorded, and at any point in time, a clear lifeline as a CI traverses through states is available.

Based on the data recorded, reports can be generated to provide an accurate representation of the CMDB ecosystem. A number of reports are generally developed as a part of the service asset and configuration management process. A few include configuration baseline and snapshots, active CIs in production, CI changes that are unauthorized, and CI changes performed in a specified period.

Step 5: Verification and Audit

The CMDB is built and used by the service management teams on a day-to-day basis for all their needs. But how do we know that the data residing in the CMDB reflects the true representation of the configuration items in production? For all we know, somebody could have made changes to some of the servers and switches without updating the changes in the CMDB. The verification and audit activity ensures that the accuracy of the CMDB is checked on a regular basis.

This activity can happen in multiple ways. Today we have automated auditing tools that monitor CIs in production and compare them with the CMDB values. If they don't match, a flag goes out to the concerned teams indicating the anomaly. This is perhaps the most efficient way to verify the accuracy of the CMDB.

Then there are CIs that cannot be audited automatically such as racks on a chassis and the server mounts. These need to be audited physically. In this case, an auditor walks into a datacenter and picks up a random set of servers and switches to be audited physically. The CMDB indicates its physical location. The auditor walks up to its physical location and checks whether the server/switch sits where it is meant to.

Physical audits are limited to geographical attributes of a CI. However, there was a time when the entire audit was done physically. An auditor would ask the administrator to log into a server and then would check for the configuration matches physically. Gone are those days, as physical checks have been replaced by remote audits. The auditor remotely connects with the administrator to audit the configurations of CIs instead of being physically present at the datacenter.

When the audit is complete, an audit report is written, highlighting the lapses and providing a list of actions aimed at improving the accuracy of the CMDB. The lapses are referred to as nonconformances (NCs). A certain period of time is given to the accountable teams to fix the lapses and to come up with a preventive measure to ensure such inaccuracies don't occur again. Anomalies between the CIs on the field and the CMDB happen mostly because of unauthorized changes and the lack of CMDB changes on authorized changes.

Why Configuration Management Is Relevant to DevOps

Configuration management is at the heart of service management and is solely responsible for the efficient resolution of incidents and to provide an effective map for identifying true business impact. DevOps is not much different from ITIL when it comes to the maintenance of products and services. The other half too, development, has plenty of dependency on configuration management, as we will find out later in this chapter.

Development is not a stand-alone activity and cannot be done in isolation. Systems talk to one another and exchange data. With configuration management in place, it makes it so much easier for developers to identify the connecting bits and develop with ease and efficiency.

In some of the development projects that I was involved with, sadly enough, did not have the luxury of a CMDB or a map that gave them an accurate representation of web services, databases, and views. So, any development done on one part of the software resulted in something else breaking on the other end. The worst part was that such defects came to the fore only during regression testing toward the later part of the development and testing lifecycle. This essentially gave a few panic moments to the development team, put a big question mark around the quality of the product, and delayed the project release.

DevOps is based on the premise of fastening the development cycle and to increase the quality of the product, apart from efficient operational activities. For faster cycles of development, something like a CMDB is absolutely essential to be able to have all the information needed.

I cannot implement DevOps processes in a project until I have a working configuration management in place—something like a CMDB that gives me a view of the application and infrastructure integrations. In a software development project, configuration management is a whole lot more than just something like a CMDB. There are other moving parts that require additional management of configurations, which I will discuss in the next section.

Configuration Management in a DevOps Sense

Is configuration management relevant at all today in the age of cloud and automation? Detractors say that everything is on the cloud; resources are allocated dynamically, and at the click of a button, environments get spun up. So, what is the need to maintain the configurations and attributes if they are subject to change in a whiff?

Yes, it is true that the way we used to stack up servers has changed; in fact, it has been transformed. We no longer depend on server teams to create virtual machines and load operating systems and the set of standard settings and applications. At a click of a button, environments get created within minutes, and the resources of the server are shared between multiple server boxes, and they are logically pooled to create an environment on the cloud. The question to ask rather than point to automation is, what configuration is going into the making of the environment? How does the one-click environment creation tool know what parameters and configurations to use in the environment? The answer is rather quite simple: configuration management based on ITIL.

Consider Figure 6-3, which depicts a simple configuration management system in the DevOps world. The environment today generally lies in the cloud, and the cloud infrastructure gives us the ability to scale and descale on demand without any physical changes done to the underlying infrastructure. Furthermore, all the applications and its dependencies are deployed and maintained automatically, who gives no scope for any uncontrolled changes to creep in. The underlying configuration management that starts with the cloud infrastructure, the codebase, the binaries, and the dependencies, among others, provide a real-time blueprint of all things configuration in a DevOps project, and this will power projects to develop freely and with flair.

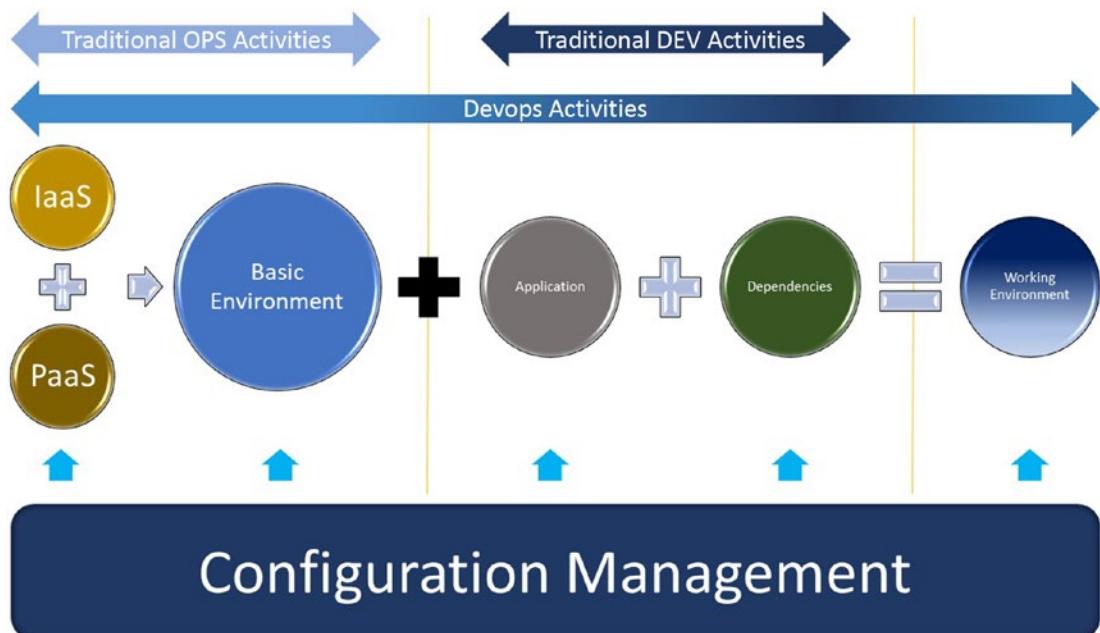


Figure 6-3. DevOps configuration management

Decoding IaaS

Traditionally building a server—which involved racking it on a chassis, hooking all the cables up, and allocating a subset of the box as a virtual machine—was the typical role of somebody working in operations. Today with the advent of the cloud and automation, the creation of a server is dynamic and can be done just by playing with a few keystrokes. The physical activities involving server racking and connecting cables are still and will always be a manual activity in a datacenter. I don't see robotics taking over this activity! However, the building of a server with specific hardware configurations is fully automated today.

To build a server, a script needs to be written with the required configurations, and the script is executed with tools such as Vagrant or Pivotal Cloud Foundry. Writing a script is a one-time activity, and it can be executed as many times as needed to create additional servers, which is generally referred to as *one-click server creation*. In fact, infrastructure-as-a-service (IaaS) providers such as AWS, Azure, and Google Cloud have created an intuitive interface to enable semitechnical people to build their own servers by playing around on their GUI.

Here's the best part about IaaS. Prior to cloud infrastructure, if we had to scale our infrastructure, we had to add physical components to the infrastructure to scale. In some cases, adding could not be done, which resulted in migration to a different piece of infrastructure altogether. Cloud infrastructure, on the other hand, is highly scalable. You need additional RAM, no problem. Just tweak the script and execute it. You got extra RAM. If you don't need additional RAM anymore, you can just as easily descale. This provides unparalleled flexibility for architects to tweak their designs and optimize for the best performance.

Decoding PaaS

After a server is built, the first order of business is to load an operating system on it. This activity was traditionally carried out by installing the operating system at the command line or via the friendly GUIs that are offered. Or, operating systems can be installed by loading preset images onto servers. Both ways work; however, they all require people from operations to do it, and since it was manual, it took a good amount of finite time.

This is where PaaS comes in to automate the process of installing the platforms (operating systems). Once the server is ready, running a script yet again with platform characteristics will enable the setup of an operating system at the snap of a finger.

We can also configure OS-level configurations such as enabling group policies, installing standard antivirus software, monitoring, and other agents as specified by the organizational policy.

PaaS is also flexible. If you want to change a configuration across a server farm, you just need to change the script and execute it. Changes to thousands of servers can happen in an instant, saving hours of manual dispensation. Tools such as Ansible, Puppet, and Chef help with installing and maintaining software configurations.

Application Deployment and Configuration

In Figure 6-3, I have indicated that the combination of IaaS and PaaS basically gives you a bare shell environment on which applications and their dependencies are loaded to complete the setup of the environments, in other words, testing and production. The application deployment was traditionally done manually, and the application was configured with its dependencies and database connections. This was traditionally the activity that the development team used to perform in testing environments and was mimicked by operations teams for production environments.

Automation has made a significant change in the way we deploy and configure applications. The tools that I discussed for use with PaaS are capable of delivering application packages along with all the necessary configurations at the click of a button.

So, does this mean that the development team's role has been trimmed to development and testing alone? In a sense, yes. We expect developers to focus on their core areas and leave the repetitive activities for tools to execute. In addition, unnecessary human errors are eliminated in the age of automation.

Underlying Configuration Management

In Figure 6-3, all the configuration activities performed have a common link: configuration management (and, more specifically, ITIL's configuration management). Let's break this down.

To perform IaaS (in other words, build a server), I need to know what configurations I need to employ. How would I get this information? I can get it from the architects or configuration management plans, if it's a new development. What if it's a system that has existed for ages and has been modified, changed, and upgraded several times over the years, and none of the blueprints carry the exact specifications of the server. In this

case, where can I get the specifications of the server? Logging into each one of them and getting the details or even running a monitoring and discovery script is not too dependable. What you need is a database that is live and that has been updated with the true configurations of the server over the years: the configuration management database. The CMDB, which is at the heart of configuration management, has all the answers that IaaS toolsets need to build the server.

Even after the server is built all the way up, the configurations still need managing. Whether the configuration data lies in the configuration management toolset or the CMDB, it does not matter. The configuration management toolset can work as a federated CMDB to ensure that the CMDB has all the data available in a single database.

Likewise, PaaS too takes configuration details from the CMDB to complete its tasks, and the application deployment and its dependencies can be effortlessly identified in a CMDB using the existing relationships between various CIs. So, the whole network of application dependencies can be brought together, stitched, and deployed to make the environment whole again with a minimum effort, ensuring maximum productivity for the involved teams.

Once everything is set up, the working environment goes into the CMDB as a CI or a bunch of CIs with relationships and get managed until its retirement.

Automation in Configuration Management

Cloud infrastructure changes to the configuration are made quite rapidly. If the configuration management is to remain updated, then it needs to be updated as rapidly as the changes itself. Yes, that is true, and that is where the automation in configuration management comes into play.

Making changes to configurations, although done quite rapidly with minimal lead time, requires governance—something like the change management process to govern the changes done to ensure that it does not disrupt services. There is a special provision in change management called *standard changes* where changes are pre-approved after a thorough assessment of business risks, which allows changes to be carried out with minimal lead time. I will talk about standard changes in detail in Chapter 9. However, for now, let's assume that changes are done on the back of standard change tickets. It is possible that we can make CMDB updates automatically based on the standard changes instantly after the change has been deemed successful. This way, the CMDB data remains accurate with the environment at all times, thanks to the automation that we have employed to make it happen.

One way to visualize this happening is that the CI where the changes are going to happen are registered in the standard change. The CI is referenced from the CMDB. So, when the change is carried out, the referenced CI gets a trigger to change its configuration. But where does this configuration data come from? It may or may not come from the change ticket. However, a more accurate source of configuration changes is the monitoring toolsets that we use to keep a finger on the pulse. These tools pick up changes to the configurations, confirm whether there is a change ticket associated with it, and if yes, then go ahead and update the CMDB directly. *Voila!* Suppose that the change carried out is an unauthorized one (no change ticket to back it up). In that case, automation helps flag the unauthorized changes to the relevant parties. In fact, automation can go one step ahead too. Based on the lack of a change ticket, the configuration toolsets like Ansible and Puppet can self-correct the configurations to undo the changes. So, thanks to automation, we can fully ensure that no unauthorized changes take place in the ecosystem. And CMDB data remains accurate and provides a true reflection of the network of CIs in the datacenter and the rest of the ecosystem.

Who Manages DevOps Configurations?

The activity involving building a server—virtual machines, group policies, and the operating system—has always been part of the operations space. The server teams were made responsible for discharging the duties related to building the server up to its operating system. In some cases, they even deployed the applications manually, tweaked the services involved, and set up applications.

As mentioned earlier, building servers is done in a completely different way today. The building of a server has been codified using scripts, and writing scripts can be seen as an activity that might fall under the developer's realm. So, it begs the question of whether the activity of building servers requires hardware engineers (read: operations).

This is a common trap that people who try to dissect DevOps fall into. We are no longer dealing with dev and ops teams anymore. We are dealing with DevOps teams, where the team works as one unit to get the job done. If the developers can write scripts and execute them to build servers, then so be it. While they build a server, they are carrying out an operational activity and not a developmental activity. The same person wears different hats while executing different activities in this case. Therefore, in a DevOps team, it is important not to differentiate people based on dev and ops but rather recognize them as a collective unit—a unit that collaborates to bring the best of both worlds into building and maintaining a product.

To conclude this topic, the DevOps team manages configurations related to the CIs that come under its scope. We don't call someone the configuration analyst or manager because they manage configurations. In fact, in a DevOps project, at a DevOps team level, we don't even like to manage configurations manually but rather automate them so that changes happen automatically and keep up with the rate of change. When there are new CIs that are introduced, a central configuration management team that is accountable for the overall design and maintenance of the CMDB will be brought into the loop for seamless additions and possible deletions.

Comprehensive Configuration Management

In DevOps, configuration management has a wider scope than a typical services organization would require. For operations to run, a services organization with a good working CMDB will be utterly satisfied as it gives them the foundational support to resolve incidents quickly and to assess the business impact accurately. For development teams to be satisfied, it's a different kettle altogether. The superset of configuration management, which is indeed what a DevOps project requires, is the comprehensive configuration management.

Comprehensive configuration management (CCM) is meant to be a single source of truth for a DevOps project. Data integrity is integral to the success of any project, and this is accomplished in DevOps through CCM, which maintains the entire product, project, and service information. To extend this concept, no information pertaining to a DevOps project must reside outside the CCM.

Comprehensive configuration management consists of three parts, as shown here (an illustrated in Figure 6-4):

- Configuration management database
- Source code repository
- Artifact repository

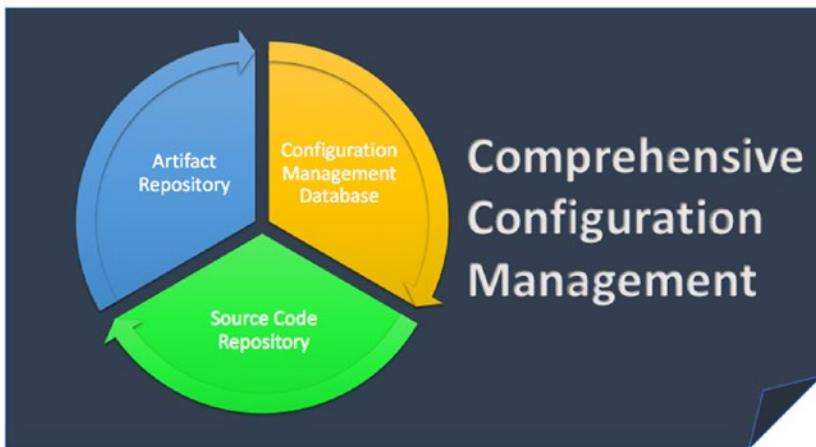


Figure 6-4. Comprehensive configuration management

The idea behind comprehensive configuration management is simple enough. The entire lifecycle of DevOps starts with development and continues to exist with operations. Therefore, the configurations for a DevOps project start with the configurations around managing source codes and the branching strategies. The binaries generated need to be managed with care and with nitpicky organizational skills. In DevOps, we recommend that developers check in the code multiple times a day, resulting in multiple builds and binaries. Not all the binaries will end up getting pushed to various environments. Therefore, it is critical to segregate the builds that get pushed to production and those that don't make the cut. Binaries are stored in an artifact repository, which is part of comprehensive configuration management.

Configuration Management Database

A CMDB comes with multiple views—one that is beneficial to operations, one for developers, and maybe another as a superset of all information such as a complete CMS view.

The developer's view of the CMDB can possibly have the various integrations of the software, including the data sources, data consumers, residing environment (servers), databases, and database instances. The operations view will possibly go deeper into the infrastructure, and the CMDB of the data sources and data consumers will have all the possible information to troubleshoot incidents and to identify root causes of problems. So, although a CMDB is a vast repository of data, it is possible to customize the views to ensure that the developer or operational personnel using it does not get drained from data overload.

A CMDB is essential for a development team's day-to-day activities, as it helps to draw a blueprint of all integrations. This will help the architect plan the development activities better, with no element of surprise coming in the latter part of the development process. It supports developers write better code considering all the possible integrations and avoids defects due to regression issues. The overall quality of the software developed improves because of fewer defects, and most importantly it avoids rework and boosts the development team's productivity.

For operations, a CMDB is like pure gold. When incidents or problems are raised against the application, it helps the team to troubleshoot the issue faster and thus reduce the downtime of the service. During the planning of enhancements as well, an accurate business impact can be drawn with a CMDB in place.

To summarize, a CMDB is an inherent part of any project, be it ITIL-driven services or DevOps-powered software development project. Building and maintaining a CMDB is an onerous task and involves a significant portion of the budget investment. However, the value gained by it outweighs the money spent building and maintaining it.

Figure 6-5 shows a CMDB. In the illustration, services, applications, databases, and servers are represented by the different colored boxes. In the illustration, Service 1 depends on Application A, as shown by the arrow relationship. Application A leverages Database 1. Both Application A and Database 1 reside on Server A.

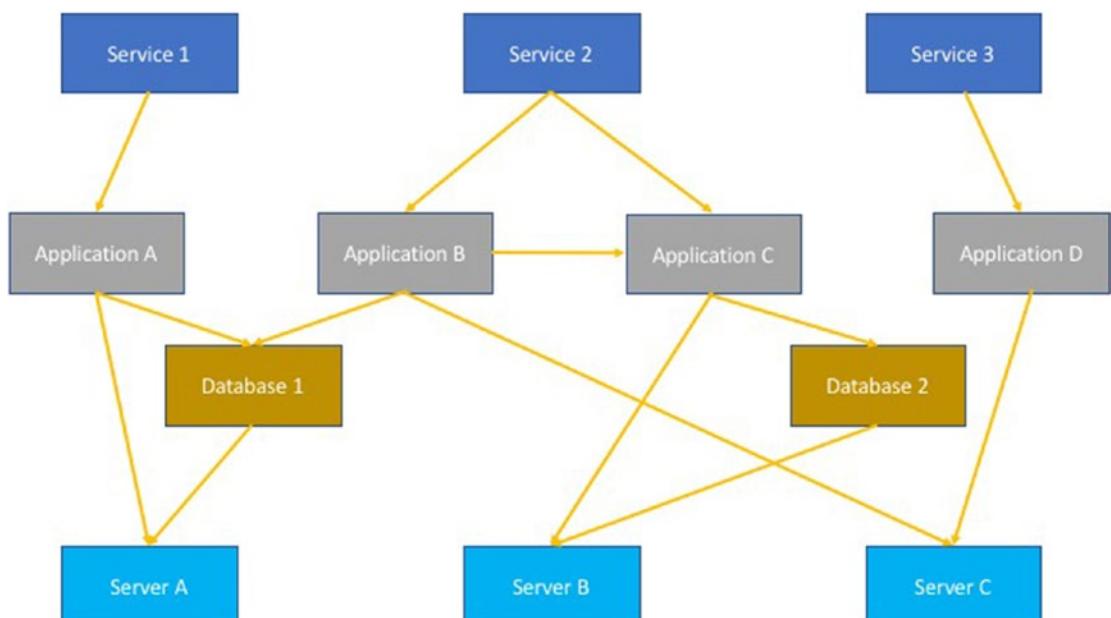


Figure 6-5. Another Illustration of a CMDB

In a real CMDB, however, the arrows mean something different from the relationships described. For example, an application generally makes use of a database; this is the relationship between the two entities. An application residing on a server leads to a dependency on the relationship. The relationship around the data flow between applications (indicated between Application B and Application C) is one of the data dependencies.

CMDB for Change Management

The CMDB is particularly useful when you are trying to make a change to any of the applications, databases, or servers.

Let's say you want to make a change to Application B. To make the change, you must first do an impact assessment. A CMDB helps in performing impact assessments, and in Figure 6-5, suppose changes are done to Application B. The impact assessment will read that any changes done to Application B will impact Application C, as the data is flowing through it.

Today, software development seldom happens in isolation. The software to be developed either is an improvement over existing code or is getting plugged into an enterprise network of applications. Therefore, it is critical that the impacts are assessed correctly, and a CMDB is a great help in this area.

CMDB for Provisioning Environments

Another application of a CMDB in DevOps is in environment provisioning. Today we can spin up environments on the go with tools such as Ansible in our scripts. When you key in the exact configuration of the production server, the environment-provisioning tools create a prod-like server with the click of a button.

But how is it that you are going to obtain the complete configuration of a server? The most straightforward way is to refer to a CMDB.

Let's say Server C is a production server that needs to be replicated. In the CMDB, the Server C entry will provide the complete configuration of the server, which is a great help in creating provisioning scripts such as with Playbooks (compatible with Ansible).

CMDB for Incident Management

The CMDB also has other benefits such as supporting the incident management teams during incident resolutions. The CMDB readily provides the architecture of applications and infrastructure, which is used to troubleshoot and identify the cause of the issue.

Source Code Repository

A source code repository (SCR) is a critical element of a DevOps project, as the entire basis for delivering software quickly starts with the organization of the source code and related scripts. The SCR has never been included or referred to in the ITIL publications because the design and maintenance of services depended on the production instance of the application rather than the means to achieving the application instance. If you are strictly from an ITIL background, then you will find that the SCR is a completely new subject that was never addressed in any of the ITIL versions and publications. The SCR is an integral part of the CCM, and to make configuration management whole again for DevOps projects, it is critical that the nuances of the SCR are well understood and included during the design and implementation of the configuration management process for DevOps projects.

Basics of a Source Code Repository

A source code repository hosts the codebase used in the development of an application. It is a version control system that allows multiple versions of the source code to be stored, retrieved, and rolled back at any point in time. This will act as an insurance against code changes that potentially could break the application. An SCR is the single source of truth for the entire project team, and it is also the medium that allows the team to collaborate and work as one unit. Source code repositories are also called *version control systems*, and the management techniques involved are referred to as *source code management*.

The objective of an SCR is to bring about a clear understanding of the constituents of different versions of a software. If you were going to release software version 4.4, then what exactly does it consist of, the contents of the release notes? To control the software and its configuration, this information is critical. In addition, every version change that was done to the SCR has a name associated with it, a timestamp, and a summary of changes performed. This showcases the evolution of a software and will come in handy during firefighting exercises.

What Can Be Stored in an SCR?

This is a million-dollar question. SCR in principle is a repository where files are stored and code changes are performed with features to allow collaboration. In a repository, typically anything and everything can be stored. This includes documents, libraries,

contracts, databases, and the source code, of course. However, the best practice is to restrict the SCR to store the codebase, build scripts, test scripts, deployment scripts, stored procedures, and configuration files.

An easy way to remember is that data that is readable by humans goes into an SCR. Anything that isn't readable goes into an artifact repository. This is just a broad principle. There could be some exceptions to this rule.

Good Practices for Achieving DevOps Objectives

A DevOps project can be deemed successful if it can accelerate the speed of delivery and reduce the defect count. For this to happen, some good practices around source code management are necessary.

In Chapter 1, when I discussed the process of continuous integration, I wrote of the need for developers to check their code in at regular intervals. This is absolutely necessary. However, even before we go into the length of delivery, it is important to lay down a hygiene factor involving storing source codes and other scripts that are stored on a source code repository. First, all source code and scripts must be stored in an SCR and nowhere else. I have seen some developers store code locally and check it into an SCR as and when needed. This is not a good practice if a team is involved (which is generally the case) in the software development. No files must be stored locally, and code changes done on a local machine must be checked into an SCR in short batches to obtain fast feedback and allow other developers to make adjustments according to the code changes, or vice versa.

An SCR is a safety net that helps you experiment. Since it is a version control system, any mistakes you make can be undone at the click of a button. In DevOps projects, experimentation is encouraged, and an SCR ensures that no matter what you screw up, you can always go back to the previous state without working your sweat glands.

Choice of an SCR Tool

There are a number of commercial off-the-shelf (COTS) and open source repositories available today. Architects who are implementing DevOps projects are spoiled for choices with differing feature sets, which makes choosing all the more complex. Before we get into the actual tools that are used today and the benefits behind them, we need to understand the architecture and principle behind these toolsets.

Source code repositories or version control systems come primarily in two different architectural standards. Traditional SCRs fall under the architectural standard called a *centralized version control system* (CVCS). The concept is straightforward. The source code is stored in a centralized repository and can be accessed by all authorized developers. At any point in time, one of the developers can check out the code, make changes, and check the changes back into the repository. This is illustrated in Figure 6-6.

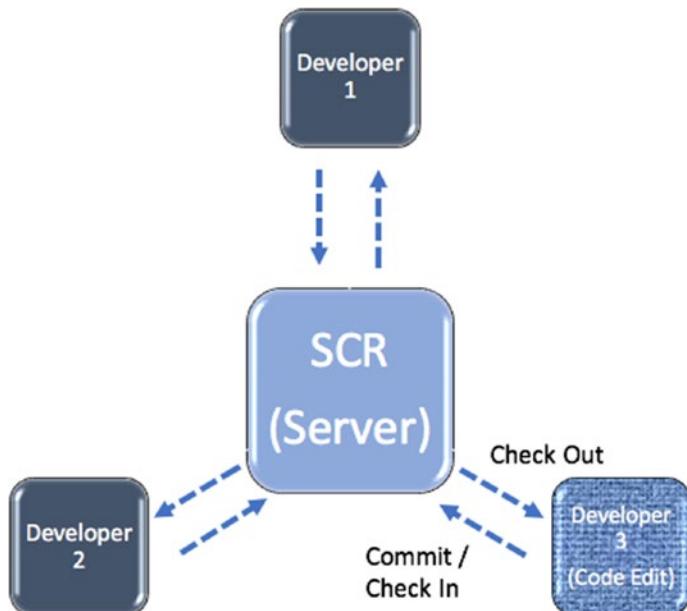


Figure 6-6. A centralized version control system

In a CVCS, the codebase is stored centrally on a server, and all developers have to access it to make changes and check it back in, something similar to the traditional file repositories such as Microsoft SharePoint. When a developer makes a change to the codebase, other developers have to pull the changes to their local systems, and the incremental changes appear in the local repositories. The positive aspect of a CVCS is that the developers are not expected to store local copies of the codebase, and as soon as they are connected to the network, they can access the server and pull the entire codebase. This is also one of the main disadvantages. Suppose the network is down or the repository is down; the developers cannot do anything but wait for the systems and networks to come back up. This affects the productivity and hence the project timelines. This shortcoming is taken care of in the modern architecture of an SCR: a distributed version control system (DVCS).

A DVCS is far more flexible as the codebase is stored locally in every developer's machine, and it gets updated directly from the server as well as from other developers, which is similar to how torrent downloads work. So, even if the network to the SCR server or the server itself is down, the developers need not worry and can carry out working on their local copies and sharing their changes dynamically with other developers. The SCR server will no longer be a single point of failure in a DVCS architecture as is the case in CVCS. This is illustrated in Figure 6-7.

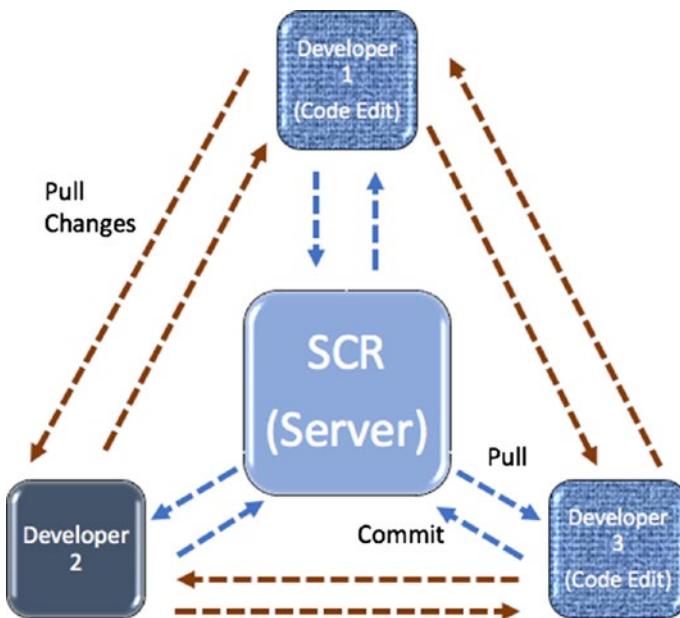


Figure 6-7. Distributed version control system

Another thing to note between the two architectures is that in DVCS you see that two developers are concurrently making changes to the codebase, while in CVCS, the codebase gets locked when a single developer checks it out. In DVCS, concurrent working is one of the major pluses and is a differentiating factor in being the SCR architecture of choice in DevOps projects.

As the developers don't depend on a server for pulling and committing their changes, the whole process of development is much faster, and it promotes collaborative working, which are sweet words to hear for any DevOps project.

Tools that run on the CVCS architecture are Subversion (SVN), CVS, and Perforce. SVN is perhaps the most popular of the CVCS toolsets and is quickly getting replaced by DVCS ones such as Git and Mercurial.

Artifact Repository

An artifact repository is a database for storing binaries primarily. In addition to binaries, you can store libraries and product and project-related documents. All machine-readable documents go into an artifact repository and not into a source code repository, primarily because artifacts are bigger in size, and the collaborative features of an SCR are overkill for dealing with binaries. Also, storing source code on SCR comes at a premium price, and you don't want to be paying a whole lot more than you have to for binaries—for the simple reason that binaries are reproducible based on the versions of codebases.

The principle of continuous integration encourages developers to push code to the mainline frequently. Each push triggers a build, which results in a binary getting generated. Depending on the size of the project, an artifact repository could end with thousands of binaries. Therefore, the management of it can be a headache if there is no proper strategy around it.

Management of Binaries

Binary management can be cumbersome; with thousands of binaries, which one should the release manager choose to push into production? Believe me, this is an undesirable task. To the release manager's aid, an artifact repository will help big time.

An artifact repository comes with two logical partitions for storing and managing binaries.

- Snapshot
- Release

Every time a build is successfully run, the binary that gets generated is stored in the Snapshot repository. But not all of the binaries get pushed into production unless the project adopts the continuous deployment methodology. The binary that gets pushed into the production is first moved into the Release partition before getting deployed into production, as illustrated in Figure 6-8.

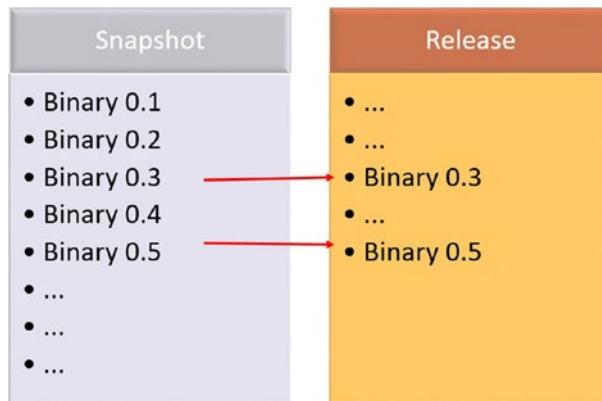


Figure 6-8. Artifact repository logical partitioning

Figure 6-8 demonstrates that various binaries are generated every time the build is successful. The binaries in this example are named Binary 0.x and are stored in the Snapshot partition. Not every binary gets promoted into production.

The binaries that are promoted get moved into the Release partition from the Snapshot partition. In this example, that includes Binary 0.3 and Binary 0.5.

This is key to the release management process. Let's say that Binary 0.5 is deployed into production, and the deployment fails. As a fallback step, the deployment must roll back to the previous version.

The previous binary versions used are stored in the Release partition, making planning and executing releases efficient.

Without a logical partition, imagine the planning and effort it would take to identify the previous version in production and roll it back.

CHAPTER 7

Incident Management Adaption

Whenever somebody refers to IT service management or ITIL, the first process that comes up on the table for discussion is the incident management process. No matter how far away one might be from the service management area, they always seem to be quite familiar with the process and its relevance. It is a highly popular process that finds its rightful place in every single organization. Since this process makes or breaks an organization's service delivery, service providers often give plenty of weight to the process, and as a result, the incident management process is perhaps the most mature of all the ITIL processes.

In this chapter, I assume that you are new to the concept of incident management, so I have written the first couple of sections to provide all the insights into the world of incident management from an ITIL perspective, which is the baseline that we are going to draw for the DevOps adaptation. If you are well versed with the incident management process, you can directly skip ahead to the DevOps adaptation sections. However, in my experience as an ITIL trainer, ITIL practitioner, and ITIL consultant, I find that many people believe that they know incident management but their understanding is further away from the ITIL's version. The rationale behind my supposition is that ITIL practitioners often believe that an incident management implementation in their respective organizations to be absolutely correct, which is not often the case. Most organizations tweak and turn the process to their advantage, and there is nothing wrong in doing that. But, when it comes to an individual's understanding of incident management, the person must draw a clear line between what the incident management baseline is and what is implemented in their organizations. With this logic, I recommend everyone read all the sections in this chapter to get a better handle on the ITIL incident management process and the adaptation of it in a DevOps environment.

What Is ITIL Incident Management?

Incident management's main aim is to reduce downtime once the service is disrupted. The process does not get into the areas of prevention and thus reduction of downtime but rather is a process that lives and breathes in the reactive realm and jumps into the ring when the service is down. Let's see some examples of services. Cable TV, Internet, and electricity are some services we subscribe to, and the expectation is that they must be available around the clock. Let's say that you come back tired from work, pick up a cold beer from a fridge, and sit down in front of TV to enjoy your time. If the cable TV service goes down, then obviously you cannot enjoy the service that you are paying for. In this instance, your service is down, which is a disruption to the service you were meant to enjoy. However, just because the service is down does not mean that incident management is at play. Perhaps your service provider does not even know that your service is down. When you lodge a complaint, an incident is logged against your cable TV account, and the incident management process gets triggered. Since you are logging the incident, it is considered to be reactive incident management as you are reacting to a reported incident. Suppose your service provider had a mechanism to monitor cable TV connections in real time and the cable TV service goes down in the afternoon. The service provider would soon enough know about the outage and can raise an incident without being called in. The process that deals with monitoring and acting on various states of monitoring is the event management process. The event management process monitors various critical points for defined changes of state, and when the change of state refers to a loss of service (or degradation), an incident gets logged automatically (capability exists), and the incident management process takes over from the event management process. This is proactive incident management where the incident is logged even before users have identified anomalies in the system.

Incident Management Is Vital

The incident management process plays a vital role in any service provider organization. It is the process that influences customer perception, it is the process that ensures that customers aren't at a disadvantage because of the lack of services, and it is the process that opens the communication channel on a regular basis with customers and keeps a lid on expectations before the steam blows all over. Therefore, the incident management process must be developed and implemented with utmost precision to ensure that the customer's sensibilities are understood and able restorers are deployed to work

when needed. This is the area where DevOps can have a momentous role to play as the integrated DevOps teams provides the incident manager with the best of resources/staff to be parachuted into an incident resolution as and when needed. This luxury of having knowledgeable and capable resource-handling incidents in a non-DevOps incident management process is generally rare.

Incident Management Is the First Line of Defense

I see incident management as the first line of defense. The service desk is the first point of contact, you might think. That's true! The service desk is a function that plays a role as first-line support. When they cannot resolve the incident, they pass it onto L2 and L3. More often than not, the incident management process will be able to find a resolution and bring the service back online—whether it involves tactics that are permanent in nature or are temporary is not of significance. At times when critical incidents emerge or when temporary resolution is applied, the problem management process picks up the slack to perform a detailed investigation and to apply a permanent solution. I will talk about problem management later in this book.

Digging Deeper into Incident Management

Incident management is a vast topic, and with the maximum number of IT service management professionals working in this process, it has grown a lot in terms of maturity and the ways of working. Although the section title says “digging deeper into incident management,” I am just going to skim the surface of the incident management process to provide a basic understanding of what the process entails.

Objectives and Principles

The main objective of the incident management process is to restore the service to its normal self as quickly as possible—the focus being on speed rather than on how it's achieved, as long as it's not disruptive. Speed is critical because the service is down, and a service that is not enjoyed by the users and customers has the potential to lead to business losses, and in turn, the service provider gets penalized based on the signed contracts.

Note Here's the definition of an *incident* from the ITIL publication: "An unplanned interruption to an IT service or reduction in the quality of an IT service."

Let me provide an example to illustrate the means of resolution over permanency. Let's say that the printer located in your bay has run out of juice and you need to make a printout for an upcoming meeting. You would register an incident, as per the standard process, but it takes about a couple of days for a new ink toner to arrive and another day for it to be installed. You don't have that kind of time on your hands, and moreover incident management frowns over delays such as this. Therefore, the incident analysts help you install a printer that is in the adjacent bay, which will enable you to make prints before the meeting. Here, by providing a temporary solution, users are unaffected for long periods of time, and the work gets done as it should seamlessly—maybe with a little blip. On the back end, new toner is ordered, and its installation is processed. It's a win-win situation. This temporary solution is referred to as a *workaround* as it does need to be tweaked again to bring in the permanency. As in, you cannot expect users to use the printer in the adjacent bay going forward—indefinitely.

To emphasize the point, the objective of incident management is to make sure that the service comes back to normal state at the earliest, no matter how the resolution is applied. To ensure that the applied fix is stable and long standing, there are other processes made responsible.

What Can Be an Incident?

Any disruption to a service is an incident. When we say *service*, we refer to services that are in active state and services that are currently available and being enjoyed by the user community. Suppose a service is getting designed and something in the process goes wrong. This cannot be an incident. Possibly it is a defect or a bug that needs to be fixed using the software lifecycle management process. The incident management process must be strictly applied to the services that are live.

Well, there are always exceptions but with a rationale. Suppose a software development team encounters an issue with the system testing environment. They go ahead and raise an incident. Why? The software developers and testers are the users (internal customers) for the testing environment service that is put into place to develop applications.

Another scenario is when the tool that monitors the health of network switches is down. You still register an incident although the customer's service is unaffected. Why? If you consider monitoring as a service, then yes, it's a service that has an internal service provider involved. Second, the lack of a monitoring tool is a risk that poses greater danger to services—delayed incident detection and hence longer duration of downtime.

Who Can Register Incidents?

So far we have discussed two instances.

- Users report incidents based on their observation or experience of service degradation or lack of service usability.
- Monitoring tools keep a close to real-time watch on the service, and as soon as it veers from the normal, an incident is automatically registered.

Users reporting incidents is highly reactive, and a fact surfaces that the service provider does not know whether the services are running as it should. It is highly ineffective in terms of maintaining service uptime, although the downtime of services is calculated generally based on the registered incidents. Monitoring agents reporting incidents is a highly recommended option and is often employed in most services. The monitoring toolsets often keep track of services or devices and employ a set of criteria for registering incidents. This system is effective because many times, incidents are identified and rectified even before the users get wind of it. I would not call it proactive as opposed to reactive when users report incidents, but in the reactive quadrant, this provides the best chance of resolving incidents at the earliest—which directly serves the purpose of the incident management process.

There are two types of monitoring services available—active monitoring and passive monitoring. *Active monitoring* refers to monitoring toolsets that monitor critical points of a service or a device and register an incident during anomalies. Examples of monitoring tools include Dynatrace, AppDynamics, Splunk, and Nagios. The second type is *passive monitoring*, where the devices such as switches and routers have a built-in capability to capture data and report it to another system that studies the data and decides to take action based on the study. Passive monitoring is often not an effective partner for the incident management process to keep downtime in check. A third-party monitoring tool (active monitoring) has a fair chance of staying outside the realm of the infrastructure, applications, and network, and report back on health accordingly.

Apart from users and monitoring tools reporting incidents, there's a third source as well. IT staff who are working toward maintaining services (infrastructure, application, and networks) are required to report incidents as and when they observe them. Like users, IT staff reporting incidents is ineffective as well, but nonetheless it is an option to keep the IT staff aligned to the overall objectives of the incident management process.

Typical Process

It is important to understand the general set of activities that are followed to achieve the incident management objectives. As I mentioned earlier in this book, ITIL is based on the value derived from various organizations, and incident management is one of the founding processes. The typical process is based on the common set of activities performed across organizations. Figure 7-1 illustrates a typical incident management process in line with the ITIL service operation publication.

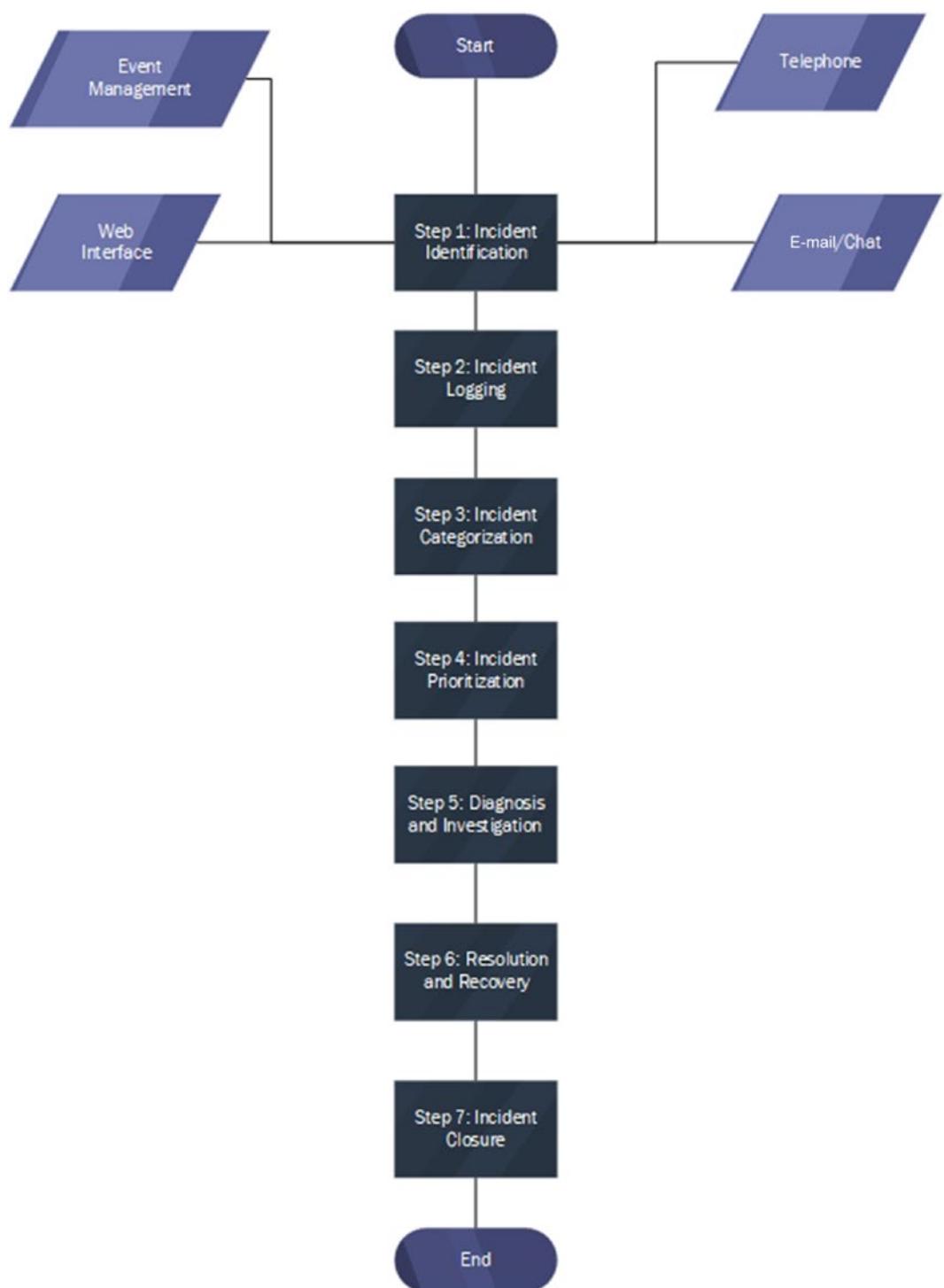


Figure 7-1. Typical incident management process

Step 1: Incident Identification

There needs to be a mechanism to identify incidents; they don't show up at the doorstep by themselves. Incident identification or triggering of incidents can happen in a number of ways. Remember that a process gets kick-started when it is fueled by the identified triggers. It is important that all triggers are identified during the process definition stage. The more the merrier, but controlling all the known triggers requires plenty of effort and could lead to misidentification of incidents if they are not well thought through. The following are the most commonly used incident management triggers:

- *Event management:* The monitoring activities that I mentioned earlier fall under the event management process, including both active and passive monitoring.
Telephone: One of the oldest forms of raising complaints is to pick up the phone and complain about a broken-down service. To raise an incident, users have the option of calling the service desk. The trigger in this case is the phone call by the users. It is also possible that IT staff could find a fault in one of the systems and call the service desk.
- *E-mail/chat:* Instead of calling in, users can opt for a passive form of communication through e-mail or real-time chat. They would still be interacting with the service desk and getting them to raise an incident on their behalf.
- *Web interface:* In today's world of cutbacks, the service desk is often replaced by self-help mechanisms. ITSM ticketing tools such as ServiceNow and BMC Remedy provide the front end for users to raise their own tickets without the aid of the service desk. In a way, it is good that precious resources can be used elsewhere. But it could also lead to a good amount of misidentified incidents that could add to the flab that you don't like to see.

Step 2: Incident Logging

All incidents that are identified should be logged, with a timestamp that is unalterable. Incidents are generally logged directly into the tool by the user if there is a web interface. And the event management tools can also create incidents based on the threshold levels and the designed algorithms. The service desk raises incidents on behalf of end users when they call, e-mail, or chat about their issues.

An incident ticket has a number of fields associated with it, primarily to support the resolution of the incident and to control the various parameters and pull reports as necessary. The following are some common fields that are found on incident tickets:

- Incident number (unique)
- End user name
- End user team name
- Incident logger name
- Time of logging the incident
- Incident medium (phone/chat/web/e-mail)
- Impact
- Urgency
- Priority
- Category
- Related CI
- Incident summary
- Incident description
- Assigned resolver group
- Assigned engineer
- Status
- Resolution code
- Time of resolution/closure

Step 3: Incident Categorization

Not all incidents fall into the same bucket. Some incidents are server based, some network, and some application/software. It is paramount to identify which bucket the incident falls under, as the incident categories determine which resolver group gets assigned to resolve it.

For example, if there is an incident logged for the loss of Internet, you need the network team in charge of handling network issues to work on it. If this incident gets categorized incorrectly, say applications, the incident will be assigned to a resolver group that specializes in software troubleshooting and code fixes. They will not be able to resolve the incident. They are required to recategorize and assign it to the right group. But the effect of wrong categorization is that the resolution takes longer, and this defeats the purpose of the incident management process. So, it is absolutely imperative that the team that is logging the incident is specialized in identifying the incident types and categorizes them correctly.

In cases of autologged incidents, event management tools are designed to select a predetermined category that does not falter. User-raised incidents are automatically categorized based on the keywords mentioned in the incident summary and description. It is quite possible that the incident could be categorized incorrectly in this scenario, but in the interests of automation, this is the price we have to pay.

Step 4: Incident Prioritization

Earlier I discussed extensively incident prioritization. This is the step where the process of incident prioritization gets acted upon. The service desk measures the urgency and impact and sets the incident priority. Event management tools have the ability to set the right priority based on an algorithm. User-created incidents are normally assigned a default priority, and the resolver group changes the priority once it begins resolving the incident. Incident priorities are not set in stone. They can be changed throughout the lifecycle of an incident. It is possible that the end user has hyped the impact of the incident and could have gotten a higher-priority incident raised. During the resolution process, the resolver group validates the impact and urgency and alters the priority as needed. Some critical incidents are monitored after resolution. The observation period could see the priority pushed down until closure.

Step 5: Diagnosis and Investigation

The service desk performs the initial diagnosis of an incident by understanding the symptoms of the incident. The service desk tries to understand exactly what is not working and then tries to take the user through some basic troubleshooting steps to resolve the incident. This is a key substep, as it provides the necessary data points for further investigation on the incident. It is analogous to a doctor asking you about the

symptoms you have: Do you have throat pain? Do you have a cough? Do you have a cold? Do you have a headache? You get the drift. Likewise, the service desk is expected to ask a series of questions to provide the necessary information to resolve the incident quickly, which is the objective of the incident management process.

Not all incidents can be resolved by the service desk. They get functionally escalated to the next level of support, generally referred to as level 2, or L2. The L2 group is normally part of an expert group, such as the server group, network group, storage group, or software group. The resolver group diagnoses the incident with the available information and, if needed, calls the user to obtain more information. It is possible that the service desk's line of questioning could be on the wrong path, and perhaps the resolver group must start all over again by asking the right set of questions. Investigation of the incident digs deeper into the incident by understanding one or more of the following thought processes:

- What is the user expecting to obtain through the incident?
- What has gone wrong?
- What are the sequence of steps that led to the incident?
- Who is impacted? Is it localized or global?
- Were there changes performed in the environment that might have upset the system?
- Are there any similar incidents logged previously? Are there any known error database (KEDB) articles available to assist?

Step 6: Resolution and Recovery

Based on the investigation, resolutions can be applied. For example, if the resolver group determines that a particular incident is not localized, there is no reason for it to resolve the incidents on the user's PC, but rather it starts troubleshooting in the server or network. Or perhaps it brings in the experts who deal with global issues. The success of resolution rides on the right path of investigation. If the doctor you are seeing prescribes the wrong medicine because the line of investigation was completely way off, the chances of recovery are close to nil, aren't they?

For incidents that are widespread in nature (affecting multiple users), once the resolution is applied, various tests have to be conducted by the resolver group to be absolutely sure that the incident has been resolved, and there is generally a recovery period to observe the incident and be on the lookout if anything were to go wrong again.

In some of the accounts that I have handled in major incident management, it was a regular practice to keep major incidents open for at least a week, to observe, and to hold daily/hourly meetings with stakeholders to check the pulse and to keep tabs on things that could go wayward.

Step 7: Incident Closure

When an incident is resolved, it is normal practice to confirm with the user before closing the incident ticket. The confirmation is generally made by the service desk, not the resolver group. So, the process for post-resolution of an incident is that the incident gets assigned to the service desk for confirmation and closure of the incident. Some organizations think that this step adds too much overhead to the service desk and prefer to forgo this confirmation. They keep the incident in resolved status for maybe three days. An e-mail is shot out to the user stating that the incident has been resolved, and if he feels otherwise, he is expected to speak up or to reopen the incident. If there is no response within three days, the incident would be auto-closed. I like doing this and have been a proponent of the auto-closure system as confirmation can be overbearing and, from a user's standpoint, irritating to the customer to receive calls just to ask for confirmation.

After an incident has been closed, a user satisfaction survey goes out asking for feedback on the timeliness of the resolution, the ease of logging incidents, and whether the user was kept informed of the incident status throughout the lifecycle.

Major Incidents

Major incidents, as the name suggests, are severely impacting incidents that have the potential to cause irreparable damage to the business. So, the ITIL service management suggests that major incidents be dealt with through a different lens. This can be done by having a separate process, a more stringent one, of course, with stricter timelines and multiple lines of communication. Many organizations institute a separate team to look into major incidents and hire those with specialized skill sets to be exposed to the pressure that this job inherits.

The people who work solely on major incidents are called *major incident managers*. They have all the privileged powers to mobilize teams and summon management representatives at any time of the day (or night). They run the show when there is a major incident and become completely accountable for the resolution of the incident. The pressure on them is immense, and it calls for nerves of steel to withstand the pressure from the customer, service provider senior management, and all other interested parties.

I once worked as a major incident manager and was heading a major incident management team not too long ago. The job entailed keeping the boat afloat at all times, and any delays from my end could potentially jeopardize the lives of miners across the globe. During a major incident, there could have been two or three phones buzzing with action, e-mails flying daggers into my inbox, and chat boxes flashing and roaring. It is a good experience when you think about it in hindsight and a time I will cherish.

To track, manage, and chase incident-related activities, there are incident managers who keep tabs on all occurrences. When a major incident hits the queue, none of the groups takes responsibility, but they call in the experts (major incident managers) to manage the situation. In some cases, the service desk and incident managers might validate the incident priority before calling the major incident line.

It is a good practice to let the whole service provider team and the customer organization know that a major incident is in progress to make sure that everybody knows that certain services are down and to avoid users calling the service desk to report on the same incident. A few good practices in this regard include sending out e-mails at the start and end of major incidents, flashing messages on office portals and on ticket logging pages, and playing an interactive voice response (IVR) message when users call the service desk.

Incident Management in DevOps

Whenever we talk of DevOps and the operations side of it, we are mostly referring to incident management and a methodology that caters to maintaining services from a downtime reduction standpoint. Making incident management work in a DevOps project is critical for the DevOps model to work. If we can get incident management right, the rest of the components in the DevOps methodology flow like molten lava.

In this section, we not only look at the process to be employed but also drill down to the management of the DevOps team—the jugglery between incidents and user stories and everything else that can make or break the incident management process within a DevOps project.

The setting before I delve deeper into the aspects of incident management in DevOps is the *one team* concept. This is a team consisting of the traditional application development and application management teams fused into a single being. It is illustrated in Figure 7-2. I am not highlighting the other roles that I did in Chapter 5 as the focus of this section is to emphasize the formation of a single team that is responsible for both development and maintenance activities. This is the team that is the alpha and the omega for the product you are supporting or the service you are delivering.

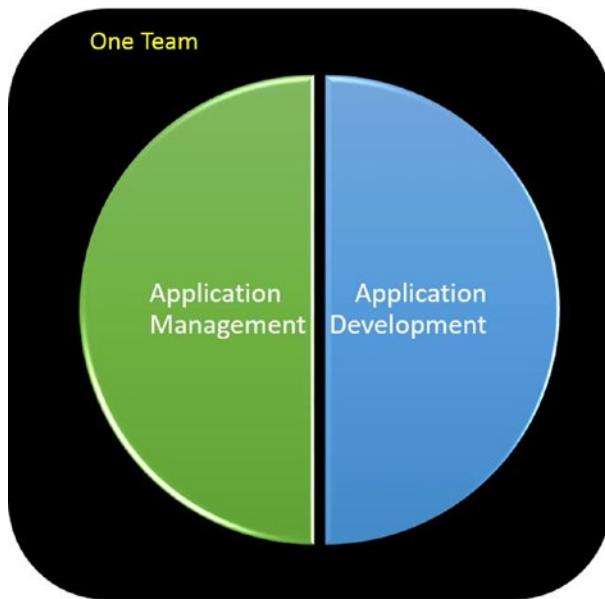


Figure 7-2. One team concept

Agile Project Management

There are multiple types of inputs for a DevOps team. It could come in the form of user stories for developing new features, or it can come through the service management pipeline in the form of incidents and problems. The DevOps team must cater to all recognized types of inputs.

User Stories

The inputs for an application development team are generally requirements from the customer. In the Agile world, we call it as user stories and not requirements. Is it the same wine in a fancy bottle? No. User stories are the goals that the user wants to achieve

from their own perspective. User stories are written in the form of users explaining what they want the software to do in various contexts. Here are some examples of simple user stories:

- I want to get suggestions when I start typing in the search box.
- I want the desktop background to refresh every hour.
- I want to see the logo as I scroll down the web site.

User stories can be little specific and more to the point to help developers by providing a context, action, and expectation. The *given-when-then* template is often used to define user stories.

- **Given** specifies a context.
- **When** provides the action that you are going to take in the defined context.
- **Then** sets the user's expectation when the action is taken in the defined context.

Here's an example:

- **Given** I sign into a shopping web site,
- **When** I type puppy in the search bar,
- **Then** a list of products with the keyword puppy must appear on a full page.

Simply put, the requirements for a development team comes in the form of user stories. The expectation from a user story is that it is a piece of work that can be completed within a sprint. But it is likely that the piece of work coming in is too complicated to be completed within a sprint cycle. These bigger pieces of work are referred to as *epics*, and they are further broken down into user stories. Development of epics can run across multiple sprints. Epics are broken down into multiple user stories, and the development of user stories is planned to be done in a single sprint.

Incidents

When there are incidents reported for a service supported by a DevOps team, the incident eventually trickles down onto the laps of the team for its resolution. The team that works on the user stories could be tasked with the responsibility to resolve incidents as well.

Problems

Although incidents and problems are mentioned in the same vein more often than not, they are a different species altogether. I have dedicated Chapter 8 to go into the depths of problem management in a DevOps project. The problem is one of the inputs to a DevOps team, and the nature of work coming from a problem is to triage on the underlying cause of complex or nagging issues. Once the root cause is known, a permanent solution is applied to avoid repeatability.

Sprints

In Agile project management methodology, we work in smaller chunks at a time, referred to as a *sprint*. A sprint could last anywhere between two to four weeks. The idea behind a sprint is not the same as what phases used to represent in the waterfall project management methodology. The activities performed in a sprint are the same as the other sprints. However, the prerequisite activities that are needed to be done before the actual development such as requirement gathering, environment setup, and environment design are done before the sprint cycle kicks in and are usually referred to as Sprint 0. The nature of the sprints is indicated in Figure 7-3.

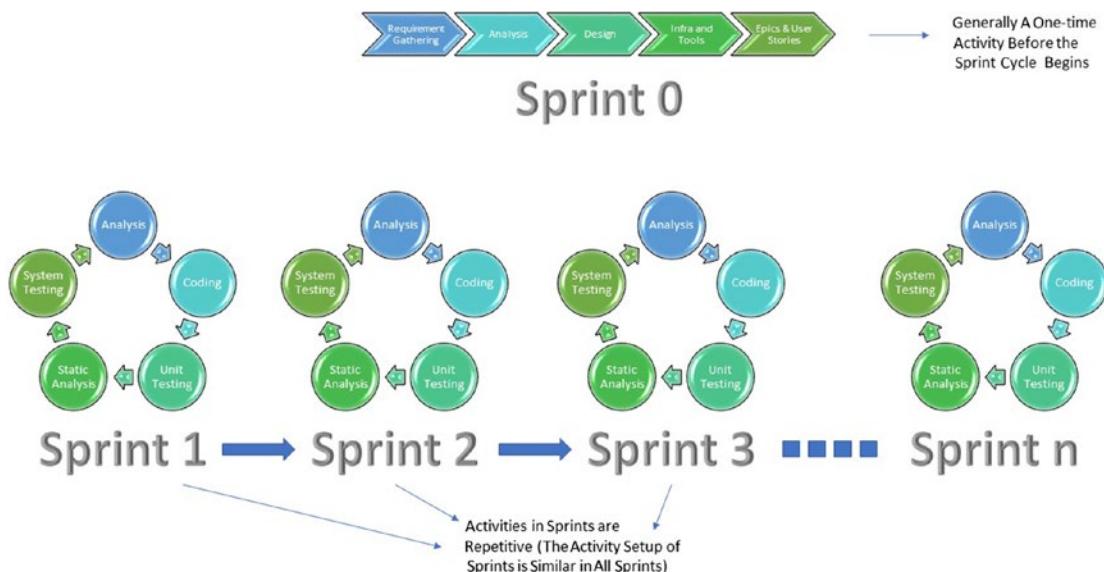


Figure 7-3. *Nature of sprints*

There can be as many sprints as needed for the development of a product. There are absolutely no restrictions. However, in the Scaled Agile framework, there is a concept of an Agile release train (ART), which supports long-running projects by clubbing the planning and execution in chunks, usually between eight to twelve weeks. Usually an ART will encompass about four to five sprints, and each ART has a planning session involving all stakeholders, governance to manage multiple sprint activities, and a review session toward the end of the cycle.

Sprint Planning

The master list of all user stories is stored in a bucket called the *product backlog*. Not all the requirements are well known before the development commences. A good chunk of the requirements get added during the project, which is the whole premise of Agile project management: to keep an open mind for changes to come midway through the project. It is expected of Agile teams to embrace changes during the development cycle with open arms, and that's precisely the reason why a holistic planning exercise is done religiously but rather planned in iterations, and the exercise is called *sprint planning*. If the sprint is two weeks, then the planning exercise will be for what can be achieved during the two weeks only, and not more.

Sprint Backlog

A subset of the product backlog is picked up in every sprint, and this bucket of user stories that is going to be worked upon in a particular sprint is called the *sprint backlog*. Based on the team's capacity, as many or as few stories are picked up.

Capacity and Velocity

A mature team with lots of years of experience can have a greater capacity compared to a similar team size with fewer years of experience. The capacity of a team is highly subjective—meaning it needs to be deciphered on the individuals and we cannot come up with a formula for identifying a team's capacity. In most cases, the capacity of a team is determined only after running a few sprints. Based on the number of user stories delivered in every sprint, a pattern emerges that indicates the efficiency of the team. This pattern is called the *velocity*. Based on the velocity, we can gauge the estimated time to complete all the user stories in the product backlog. Velocity is one of the key metrics used in Agile project management.

Determining Complexity

Not all user stories are similar in complexity. Some user stories require a few hours of development, while others may need a few days' efforts. Calculating the velocity based on the number of user stories does not accurately reflect on the team's performance. Therefore, we measure the complexity of user stories with story points. Story points are an abstract unit of measurement for user stories that loosely indicates their complexity, the duration to complete development and testing, and the dependencies involved. Every user story is associated with certain story points, and the process of associating user stories with story points is called *story point estimation*.

Estimation Technique: Planning Poker

The estimation process is not a straightforward activity either. There is no formula to help with it. Story points are determined based on relative comparison with other user stories. For example, creating a login page can be estimated to be one story point in project A where a blogging engine is getting built. But on another project involving the same login page for an Internet banking site, the login page user story could have three story points. The estimated story points for user stories depend solely on the complexity of the other story points. There are several ways of coming up with the estimation. The most popular one is through the game of planning poker.

The game of poker is played with each team member given a set of playing cards bearing numerical values—usually the Fibonacci series (1,2,3,5,8,13,21...). The user stories in the product backlog are laid out to identify the simplest and least complex user story, and this user story is given one story point. Based on this user story's complexity, the other user stories are measured. If the login page is one story point, how complex is the summary page where various data sources have to be called in? Say that it is five times as complex, so five story points are given to it.

How do you determine whether it is five times more complex or eight times more complex? Who can determine this? This is where the game of planning poker comes in handy. The team is seated around the table along with the product owner, each with their own set of Fibonacci-numbered cards. First they all agree on the simplest user story. Conflicts within team members on what constitutes the simplest work story is sorted out through discussion. In case there is no consensus, it is put to vote, and the user stories with maximum backing will be considered the simplest user story. The simplest user story is given a weight of one story point.

The product owner then goes on to pick the next user story from the product backlog and explains what is expected, which provides a fair idea to the team on its complexity. Now each of the team members measures the user story against the simplest user story. If the simplest user story has a complexity of X and if the user story that is being considered for estimation is five times as complex (5X), then the playing card number bearing numerical value 5 is drawn. Each team member pulls the playing card based on their perception and keeps it face down on the table. When everybody has drawn a card, the card is turned up, and discussions begin to deliberate the complexity and to arrive at a consensus on its complexity and the associated story points. It is likely that a developer might select three story points while another selects five story points. Each of the developers has to provide a rationale for their complexity estimation; this can either compel the other team members to change their vote or stick to their points of view. This exercise might seem to take time, but truly underneath, it is beginning a conversation around the user stories, and it is bringing the team together in arriving at a decision. It is like a jury that deliberates on a verdict hoping to arrive at a consensus.

DOR and DOD

There are several user stories in the product backlog. How do you know which ones to pick? Well, as I mention earlier, the product owner is the only person who can call the shots and prioritize requirements. However, what is the acceptance criteria for a user story to be picked up into a sprint? We call this minimum criteria the *definition of ready* (DOR), which is nothing but having all the ducks in a row in a perfect order before being considered for development.

How do you know that the user story that is picked up in a sprint is delivered? What is the agreement on completion of a user story? This common agreement on completion of development is called the *definition of done* (DOD). DOD refers exactly to the series of actions that are undertaken before signaling it as done. For example, in a project, the DOD could be as follows:

Analysis ► Coding ► Unit Testing ► Static Analysis ► System Testing

So whenever a user story goes through this cycle successfully, it is considered as done. But for another project, the DOD could be this:

Analysis ► Coding ► Unit Testing

in this case, the team is expected to run unit tests successfully before the user story is considered as delivered.

In both the examples, there is no right or wrong definitions of done. It is an agreement between the customer and the software development organization to agree on what constitutes done and to stick with it throughout the lifecycle of the product.

Sprint Planning for a DevOps Team

The sprint planning session is in play. We have identified that the inputs are user stories, incidents, and problems. So, all the three inputs make up the product backlog, and it is represented as an input funnel, as shown in Figure 7-4.

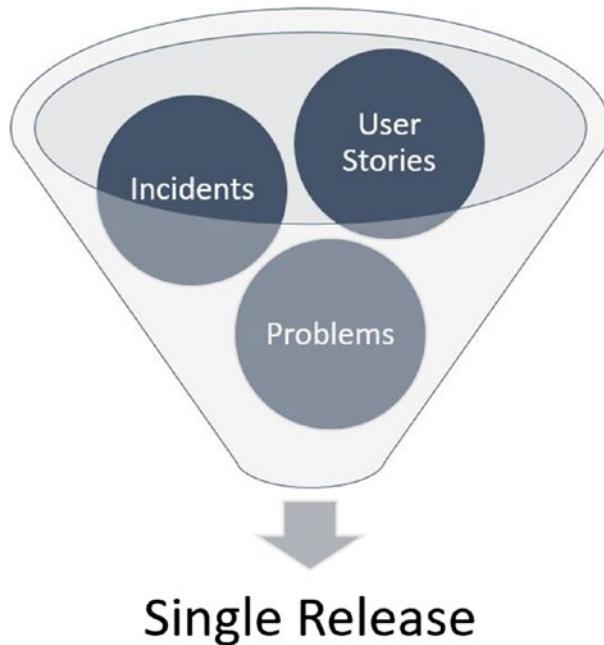


Figure 7-4. Input funnel for a DevOps team

Incidents and problems have a SLA associated with them; delays in resolving incidents could lead to SLA breaches and penalties. So, during the planning session, it might be prudent to pick up incidents that are closer to breaching. If I have four incidents, two breaching their SLAs in the next three weeks and the other two breaching in about five weeks, I will probably pick the two incidents that are breaching the SLA in three weeks.

Let's say that the team capacity is 40 story points. I have identified the two incidents that need to be resolved during the sprint. The incident resolution's complexity is identified using story points. In this case, let's say that it amounts to four story points. So, we have another 36 story points for problems and development activities.

There is one problem in the product backlog whose SLA is nearing the breaching levels, so we have decided to pick it up as well. The problem has a weightage of ten story points. So, the team has about 26 story points left for the delivery of user stories.

The team then goes on to pick user stories amounting to 26 user stories or less. One of the working principles in planning is that the identified load never exceeds the team capacity. This means that the team cannot choose more than 26 user stories. In other words, when the capacity is 40 story points, the load must be equal to less than 40 story points.

But wait. What if a high-priority incident comes calling during a sprint that requires immediate resolution? The SLA is for a few hours rather than weeks. How do we plan for such circumstances?

There are two schools of thought for such scenarios.

- Plan for what is currently on your plate.
- Keep some contingency aside during the planning session.

Plan for What Is Currently on Your Plate

You have two incidents and one problem that needs resolution in the sprint, so take it up. There are user stories prioritized by the product owner. Pick it up. If an incident comes midway through the sprint, prioritize the incident that is high priority. One or more user stories could be the casualties, but they get considered in the next sprint.

In this approach, the planning is done for what is on the plate, and incidents that come midway railroad the execution of some parts of the delivery, which is deemed okay considering that incidents pertain to the disruption of services. But in terms of identifying the team's performance, team morale could take a hit. You planned for 39 story points in a sprint, but you delivered only 26 because of the midway incident. It kind of throws the sprint and planning off-balance. But, I have seen this kind of a setup used in organizations, and it does fairly well if the product is quite stable.

Keep some Contingency Aside During the Planning Session

In the second method, a certain amount of contingency is kept aside during the planning session for midway incidents. For example, during the sprint planning session, I consider that the planning is to be done for 35 story points only instead of the 40. The remaining five story points are a contingency measure for incidents that require immediate assistance.

I throw my weight behind this technique rather than the first because it is more pragmatic than the first. In my experience, I have seen that with operations, nothing goes precisely as planned. There's always something coming back to you, either for code changes or to answer certain queries. It is good to keep some contingencies to ensure that the operational side of things is given equal priority. Moreover, even the developmental activities can benefit from some amount of contingency. After the DOD, the incremental delivery of the product could go in for a performance test, a security test, or even an acceptance test. The feedback from any of these tests can mean additional development in future sprints. A decent amount of contingency will help in taking this a long way.

How much contingency should you keep? The answer is in the metrics. Do an analysis of all the high-priority incidents that you have received for the product for the past year. Identify the frequency of occurrence and the average complexity of work that is involved. Marrying the two will give you a contingency measure to consider during planning sessions. Do a similar analysis for development activities outside the purview of sprints. Measure the frequency of feedback and the average amount of rework that is needed. A combination of rework and incident expectancy will give a good idea of the contingency to consider.

Scope of DevOps Team in Incident Management

Incidents come in all shapes and sizes. Not all are the same, and most importantly, not all incidents need to be handled by the DevOps teams. Normally in operations, we call different levels of support L1, L2, L3, and L4 (and so on). The various levels define the complexity and also indicate the team responsible for resolution.

Levels of Support

To keep things simple, I will consider only three levels, L1, L2, and L3. L1 defines basic support such as answering user queries, scheduling batches, and taking backups, among others, that are performed through an administrative page. To carry out these activities, no coding experience is necessary. In fact, the only experience that is technically needed is to follow the instructions exactly. On the soft skills front, there's plenty more to address as the service desk is usually responsible for the L1 type of support.

When an incident comes that is identified at a higher level than what the service desk is capable of doing, the service desk immediately transfers the ticket to the next line of support, L2, without trying to get their hands dirty and while trying to be brave and courageous and going the extra mile to make the customer happy. By trying things in support, we invariably waste time, which translates to an extension of downtime and the customer not being too glad to do business with you. So, as soon as the incident complexity is identified, it gets pushed to the next level.

The second level of support, L2, is one over the service desk. This team provides a configuration level of changes to the product and also involves incidents pertaining to IT infrastructure and connectivity. The teams that possibly are best suited to handle the configuration level and infrastructure changes are shared teams. These shared teams work across products and services, and they are best placed to support at a L2 level.

The next level of support, L3, involves code changes. In our case, it includes architectural changes as well. Usually architectural changes are referred as L4. So, any code changes to be effected can be done only by the L3 support team. The L3 support team has access to the code base and the necessary skillsets and setup to make changes to the code. Remember that making changes to the code is like a surgeon incising your skin to access your organs. Not all doctors can do it nor do they have an environment or setup to perform such operations. Your surgeon is like L3 support, a general physician like your L1 and a specialist like a L2 support. The L3 support or the L3 team is the DevOps team, in other words, the blended team that does both application development and application management.

Figure 7-5 the different levels of support, L1, L2, and L3 are illustrated in. There is a reason why I have used an inverted pyramid to indicate the levels. The area under the L1, L2, and L3 levels of support in the figure is directly proportional to the volume of incidents handled by each of the support teams. In my experience, most of the incidents that come through require basic knowledge for resolution, L1 by service desk. Among the ones left over, a lion's share of the incidents are resolved through configurational changes. Only a small percentage of incidents actually flow through to the L3 team, the DevOps team.

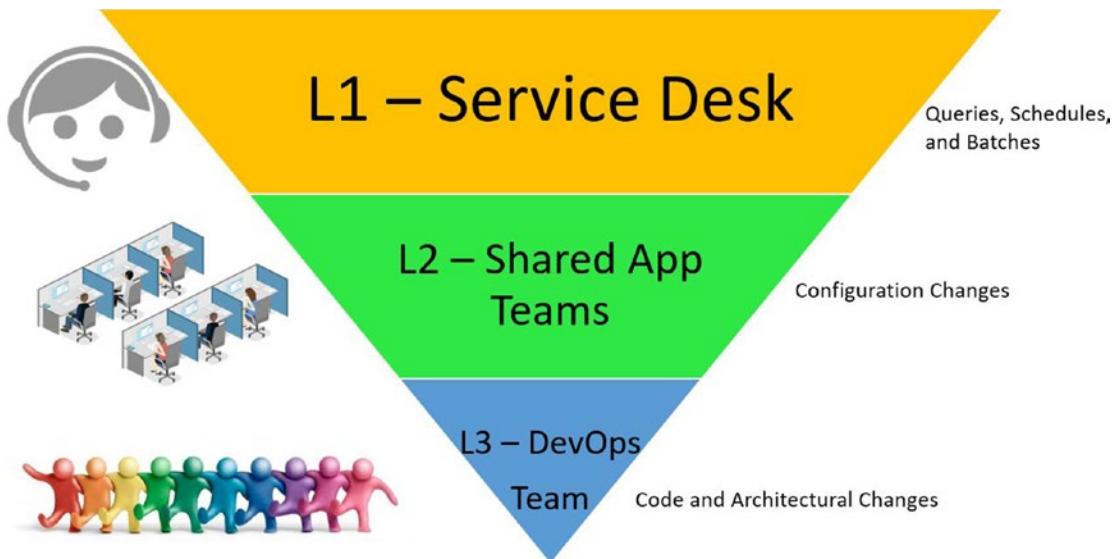


Figure 7-5. L1, L2, and L3 support structure

An incident that requires code support flows into the service desk. The service desk identifies that the incident pertains to code changes. Do they transfer the ticket to the DevOps team? The answer is no. The service desk does not and cannot make a decision on an incident requiring changes to the code. They don't have the necessary skillsets to make that decision. Therefore, their only functional escalation is to the L2 team. The L2 team, the technical management function as per the ITIL publication, is more technical in nature and is in a better situation to recommend code changes. And they transfer incidents to the DevOps team.

Does the DevOps team accept all incidents that come into their bucket? The answer is once again no.

The incident manager comes into the picture when incidents hit the DevOps incident bucket. They review the incident, analyze it, and determine whether the incident requires code changes or whether it can be resolved through changes in the configuration files. If it's the latter, the incident is pushed back to the L2 support team, and if it requires changes to the code, the incident manager gets the pertinent information (to meet the DOR) to populate the product backlog with the incident.

Incident Flow

From my experience, about 60 percent of the incidents get resolved either at the service desk or through the usage of tools that can resolve incidents automatically without human intervention. About 30 to 35 percent of the incidents come through for the configuration level of changes. And only a small percentage of incidents flow through to the DevOps team, requiring changes to the code. Remember that the code usually does not break services. There are plenty of layers above the code build such as configuration, infrastructure, and network that are responsible for causing incidents.

Note It is a good practice to pass the configuration changes through the CI-CD pipeline to ensure that the development and test environments are similar to the production environments and also to identify regression issues if any. Also, changes performed by L2 should be reviewed by L3 teams during sprint retrospectives to reflect on the changes and to identify improvement opportunities if any.

I was involved in a couple of projects where the L2 support was clubbed with the DevOps team. With L2 in DevOps team, the team had to resolve more incidents than the actual development work. The morale of the team went from being quite high to terribly low because most of the time they were consigned to working on configurational changes rather than the actual work that coders do. After this experience, it was pretty clear to me that L2 must stay outside of a DevOps team.

Knowledge at the Core

One important key ingredient that gets missed out time and again is the knowledge around the product, the history, and all that is necessary to maintain and upgrade. It is in fact as important as the configuration management, but it is often given a secondary life, with the key phrase being *if there is time*. The development is given the priority and so is everything else that contributes directly to the service or the product. When it comes to creating and maintaining knowledge, people just don't have the time. And when an incident comes calling, the developers sit for a detailed session of analysis trying to identify all the dependencies and the logic behind the application design. This analysis time eventually eats into the incident resolution time, extending the downtime and failing the objective of the incident management process.

If the intent is to make incident management effective and efficient, then start with the knowledge management along with the configuration management. Time and again I have seen projects not even maintain the bare necessities such as design documents. The remedy to the problem starts with the governance. Agile project management preaches minimum documentation with an emphasis on developing a working software. So, what the minimum documentation should look like needs to come from the project governance. For a DevOps project that covers both development and operations, the documentation around the product/service must be specific, easy to retrieve, and regularly updated.

ITIL's Knowledge Management

In fact, ITIL's knowledge management process is quite powerful and has the ability to provide credence to the management of knowledge in a DevOps project. The process exists to ensure that the service delivery is done in an efficient manner and the knowledge is available as and when it is needed. The whole premise is around ensuring knowledge exists to support the process and not that the knowledge needs to be there because we created something new.

I have heard from a few project managers that the whole idea of maintaining knowledge is done so that the organizations get ISO certified. They say that we maintain knowledge for the sake of it. This statement is so untrue. Let me peel it open like an onion. The ISO certifications are based on standards that have shown results and that are widely accepted. The controls within the ISOs are a culmination of what the industry follows as relevant and something that is absolutely necessary for the project to perform. Consider the example of a risk register, a document where you record project risks along with its respective owners, mitigation plans, and other pertinent details. You tend to revisit the document as and when needed, especially after major changes. Let me remind you that risks stand in the way of project success and failure, and maintaining one will help you understand the project risks and be prepared for them when they materialize. It is better to be ready than be surprised.

What Knowledge to Maintain

Every project is different, and every project caters to different domains. The technology is different, which changes the composition of the documents that one maintains for a projects.

Generally speaking, in a development project, we maintain a number of documents including the following:

- Contracts signed with the customer
- High-level requirements
- Design and analysis documents
- Test strategy and test plan
- Project and release plans
- Financial plans and tracker
- Estimation schemes and trackers
- Balanced scorecards
- Release notes
- Training and support documents

From a support perspective, the following are the documents that are generally maintained:

- Service level agreement
- Support documents with cheat sheets
- Knowledgebase of issues encountered
- Root cause analysis and proactive measures undertaken
- Improvements implemented
- Reports

To me, the most critical documentation that you can maintain in a DevOps project is the code itself. No, I am not talking of the comments within the code but the way the code is written. Just like how this book is divided into chapters and subheadings, well-written code can also be placed under chapters covering the various functionalities it is trying to achieve. The marks of well-written code are the following:

- *Simplicity*: Do not introduce complex loops, but rather keep the code simple.
- *Readability*: Anybody (other coders) reading the code must be in a position to understand what it is indeed doing.

- *Maintainability:* It should be easy to make changes and debug.
- *Efficiency:* If you can convey a message with fewer words, then that's the way to do it; likewise, the logic that can be realized with fewer lines of code is efficient (tools like SonarQube help in this regard).
- *Clarity:* Code must tell the story, and you must put in all the efforts for the story to reveal itself by using meaningful class and method names.

Knowledge Storing and Retrieval

Creating documentation is one part of the story; it is like a story that develops nicely, and everything solely depends on the climax. If the developed knowledge is not easily retrievable, then it's not the climax everybody is waiting for. What is the point of creating knowledge if nobody knows how to retrieve it? What good is a story if the ending is shoddy?

I have seen organizations use certain folder structures on Microsoft SharePoint and other file repositories. When it is first set up, everything is hunky dory, and after a period of time, disuse and laziness sets in. The folder meant for placing release notes stays stagnant, while release notes are stored on local drives and e-mails. The document tracker remains unread and unmodified until an audit is announced. Why do you think this problem exists?

To me this is a problem that we create. The structure with files, file updates, check-in, checkout, and sync issues are all adding to the problem. It is so complex that developers would rather analyze from the beginning rather than search in the so-called knowledge base.

What does work well is an integrated system with everything stored in the same system. You don't have the hassle of switching applications, and everything can be done with one touch. Lovely, isn't it? If you are using Jira as your project management tool, imagine a file repository built into Jira, where files can be stored directly against the design task or the user stories. Life would be so much simpler. Creating and storing documents is a whole lot of fun, and retrieval is super friendly as well. The problem of nonmaintenance does not arise as everything is right in front of your eyes. By the way,

you can get this functionality on Jira through an add-on from the Atlassian Marketplace, but it is not as good as the real thing. I am comparing it against ServiceNow, the service management tool. In ServiceNow, a separate Knowledge Management module exists where articles can be created or files uploaded just like how you would do it on a weblog. The best part is the integration with the rest of the ServiceNow system. When you create an incident, you key in an incident summary providing a brief of the issue faced. Based on the keywords, the system automatically searches the knowledge base and displays relevant knowledge base articles. Analysts and technicians working on incidents have access to knowledge readily at their fingertips, and they didn't even have to search for it. Now this makes a big difference in utilizing the maximum power of knowledge management.

The DevOps Incident Management Process

Earlier in the chapter, we looked at a typical incident management process based on industry best practices. I also mentioned that the incident management process is generally adapted for organizations implementing them based on their services, stakeholders, and comforts. For a DevOps project, the typical incident management process does go through some adaptions, but overall it remains the same in spirit and in principle. With the DevOps incident management process, we are not going to deviate too much from the normal, a sign that is good news for existing application management projects looking to go the DevOps way.

The DevOps incident management process is illustrated in Figure 7-6. The sequence of activities is indicated with seven-pointed stars.

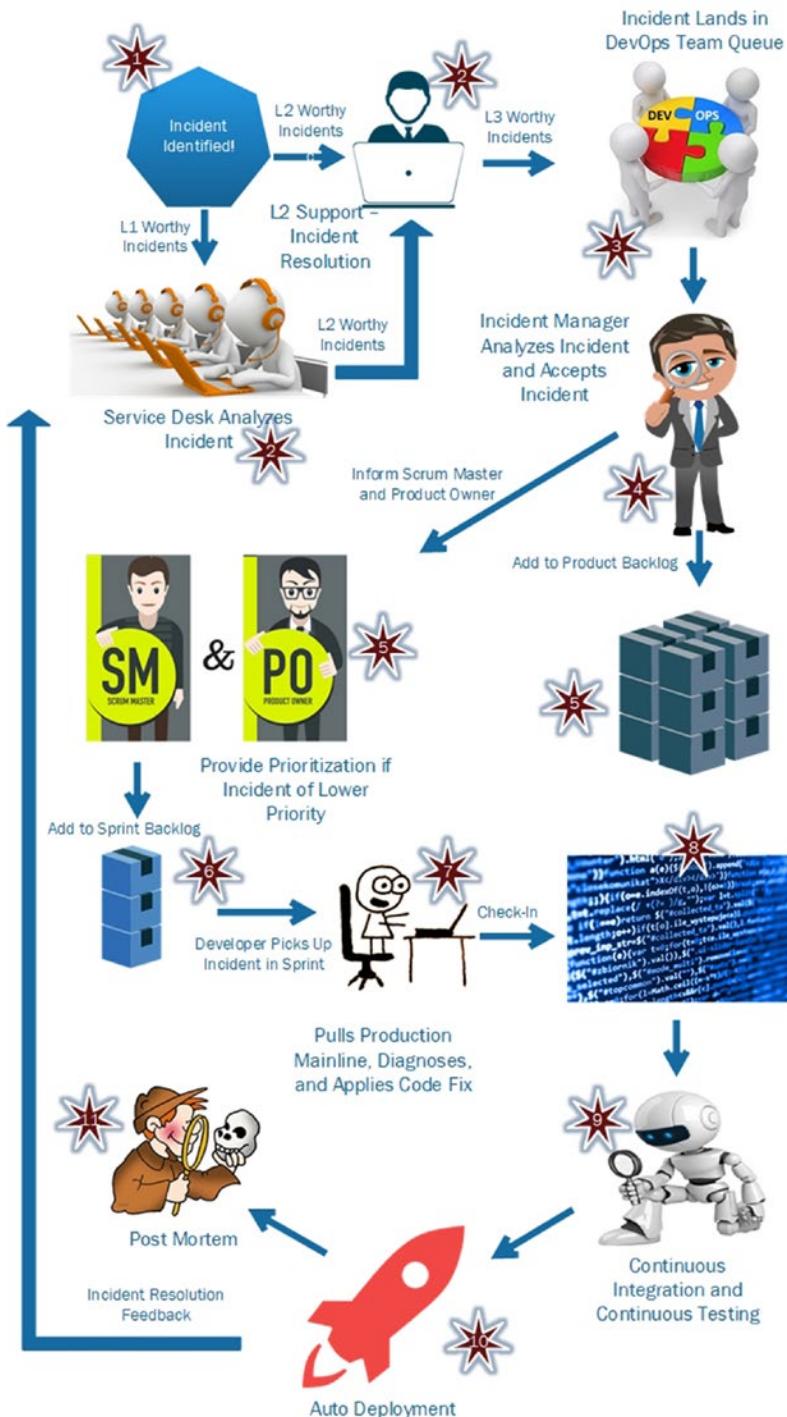


Figure 7-6. DevOps incident management process

Step 1: Incident Identification

The incident identification activity does not differ from its typical incident management counterpart. Incidents can be identified in a number of ways, automatically through monitoring tools or manually from users and IT staff.

Incidents recorded manually happens through the service desk, where the service desk prioritizes and categorizes incidents. In fact, most manually recorded incidents are routed to the service desk. For incidents registered automatically through monitoring tools, intelligence can be built in to route incidents that require specialist skillsets to L2 support directly, bypassing the service desk. This helps reduce the downtime as a better abled team is directly put into action.

Step 2: Incident Analysis, Escalation, and Resolution

There are four scenarios in step 2.

- *Scenario 1:* Incidents routed to the service desk are analyzed by the service desk, and whatever comes under their ambit is resolved.
- *Scenario 2:* Incidents that cannot be resolved by the service typically because they require specialist skillsets are transferred to L2 support.
- *Scenario 3:* Incidents that are routed directly from monitoring tools are prioritized and categorized based on embedded logic in the service management toolset. These incidents along with the incidents escalated from the service desk land in L2 support's queue. L2 support analyzes the incidents and provides resolutions.
- *Scenario 4:* L2 support cannot resolve all incidents. Remember that they are a specialist group, but their specialty does not involve making code modifications. The incidents that cannot be resolved by L2 support, as well as the incidents that require code modifications, are escalated to L3 support, the DevOps team.

Step 3: Incident with DevOps Team

This is the step where things get interesting compared to a normal service management practice. Usually there is a specialist L3 support team, the application management team that exists only to manage the services such as the resolution of incidents requiring code modifications. In the DevOps incident management process, the incident has been passed onto the DevOps team, which is primarily a development team.

Try to imagine the amount of specialty that the DevOps team brings to the table for incident resolution. The developers of the software have an intimate knowledge of the software, and this will bring down the downtime rapidly and ensure that the incidents are resolved as quickly as possible, meeting the objective of the ITIL incident management process.

Step 4: Incident Manager Analyzes and Accepts Incidents

As per the roles and structures that we discussed in Chapter 5, the incident manager is part of the DevOps team. However, if the DevOps team is fairly small without a full-time workload for an incident manager, the person can be placed in a shared role as well.

The incident manager is well aware of the product that is being developed and managed. Generally, in ITIL, incident managers are reactive folks, and they act on the incident when it comes in and, during the course of the incident, understand the intricacies of the product. However, in the DevOps incident management process, incident managers are part of the DevOps team that works with the developers. So, they are well versed with the product, and this expanded knowledge gives them a better handle on incidents that could possibly come in. This knowledge gives them the power to manage incidents throughout the lifecycle with precision and can help direct and manage multiple teams to align toward incident resolution swiftly and with a purpose.

All incidents that come to the DevOps team go through the desk of the incident manager. This person analyzes the incident, and after confirming that it requires coding-specific skillsets, the incident is accepted. If the incident manager decides that the incident can be resolved by the L2 support and does not require changes to the code, then the incident manager has the liberty to send the incident to L2 support. In certain cases, if the L2 support is unable to resolve the incident even if the incident does not call for changes to the code, the incident manager accepts the incident in the DevOps team queue.

Accepting an incident involves two things.

- Add the incident to the product backlog. Generally the service management tool for registering and tracking incidents and the product backlog tool are separate toolsets. For example, ServiceNow is a service management tool for registering and tracking incidents, and Jira is a tool for managing the product backlog. The DevOps team generally does not work on the service management tool, as the product backlog is their single source of truth. Therefore, the incident

details need to be moved into the product backlog. This can happen through a connector (B2B bridge). In this case, Atlassian Marketplace is one of the providers for connectors between ServiceNow and Jira. The incident that is accepted by the incident manager gets moved over to the product backlog with a click of a button. If no such connector exists, then the incident manager might have to manually register the incident in the product backlog.

- After accepting the incident, the incident manager must inform the product owner and the Scrum master regarding the addition of the incident in the product backlog. The Scrum master or the product owner do not decide whether an incident can make it to the product backlog or not; that decision lies with the incident manager. However, both the parties need to be notified as they need to start preparing and planning for incident resolution.

Steps 5 and 6: Incident Prioritized and Added to Sprint

The Scrum master is responsible for adding incidents to the sprint backlog. Before doing that, the product owner will analyze the incident and provide a decision on its prioritization. For example, if the incident is of low priority with an SLA of 25 business days, then the product owner might prioritize it lower for it to be accommodated in the upcoming sprint and not be included in the ongoing one. If the incident is a major incident, with an SLA counted in hours rather than days, then the product owner will prioritize the incident over all others, and it will be included in the current sprint.

The Scrum master will accordingly add the incident in the sprint backlog depending on the priority. If a high-priority incident pops up, then the incident is immediately added to the sprint backlog. If a lower-priority incident comes in, then it can wait for the next sprint or the sprint after depending on the SLAs attached to it and the prioritization of the product owner. When the incident is added to the sprint backlog, the Scrum master engages the Scrum team and the incident manager to resolve the incident. If it is a major incident, the priority is made known, and the best developers or a group of developers get into action for speedy resolution.

Steps 7 and 8: Scrum Team Makes Code Changes and Checks in

When the sprint backlog is populated with the incident (and only then), the Scrum team will start working on it. In most instances, incidents are added midway through the sprint, which is usually not a usual practice in any other development project involving the Scrum methodology. However, in DevOps, this is accepted and valid. To make way for the incident, there is a certain amount of contingency added during the planning sessions. During stable periods and change freeze windows, a minimal amount of contingency is added during sprints. When releases are being deployed into production or whenever partner systems are being modified, there is a certain expectation of regression hitting the project, which allows for additional efforts going into the contingency bucket.

The developer is usually given a brief by the incident manager on the history and facts around the incident, which might help the developer re-create the issue and could possibly help in faster resolution.

The developer might be working on a development branch, but for the incident, the developer will pull the production codebase (mainline), and whatever modifications done to it will be done on a separate branch and not on the development branch.

Let's consider the illustration of a mainline featuring release 1.3 in Figure 7-7. The development for the next release (R 2) is carried out on a development branch. All developers are working on the development branch. Midway through the sprint, an incident lands in the sprint backlog, and one of the developers picks up the incident from the backlog.

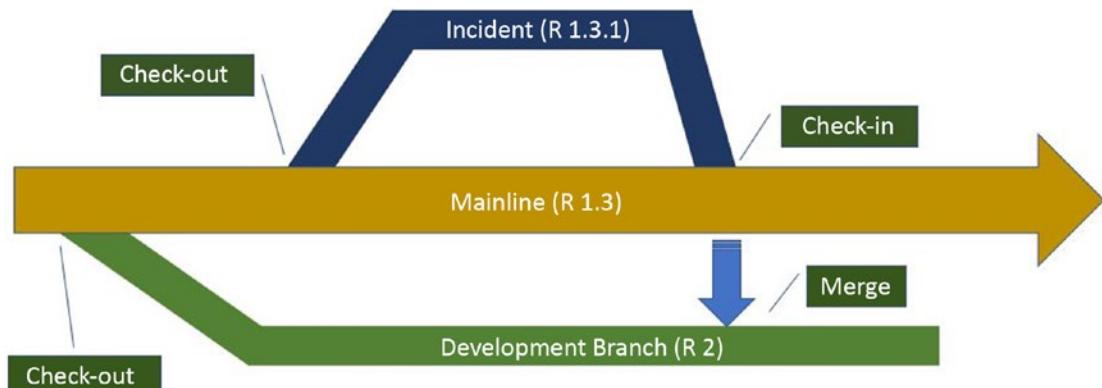


Figure 7-7. Development and incident resolution in parallel

The developer will make changes based on the production code base, such as R 1.3 on a separate branch (R 1.3.1), and not on the development branch. When the code changes are done, the code is checked back into the production mainline, considering that the incident needs to be resolved at the earliest. Let's consider that the code changes performed go through the testing phase successfully and get deployed without any issues. The developers working on the development branch will now have to merge the code changes performed in R 1.3.1 (incident) onto the development branch before continuing their coding activities. This is an important step to ensure that the incident fixes done (code changes) between R 1.3 and R 2 don't get missed when R 2 is deployed. Second, there is a probability that some of the code changes performed for the incident fix could lead to merge conflicts and will need to be resolved before build and test activities.

Step 9: Continuous Integration and Continuous Testing

In Chapter 1, I explained the continuous integration process and the concept of continuous testing in a fair amount of detail. At this point in the process, the developer checks in the code, and the automation built into the development system kicks off the build activity followed by unit testing, static analysis, and other activities pertinent to continuous integration.

After a successful build and code reviews, the binary goes through a rigorous testing process. In DevOps, we hoot for continuous testing where various tests such as system, integration, regression, performance, and other types of testing are done automatically, generally in a sequential manner. For this to be happen, the various test scripts must be ready before the code check-in. This requires a certain amount of maturity in terms of DevOps processes for the project team. While this capability is built, project teams opt for automation testing where the test triggers are done manually when the test scripts are ready. Automation testing does not enforce test scripts to be in place before the code check-in.

Step 10: Auto Deployment

Most DevOps-savvy organizations have auto deployment in place that deploys the package at a click of a button.

In the incident management process, it is important to tweak the release policies to support a reduction of downtimes. The release policy must be flexible enough to allow deployments without too many riders and outside the schedule of releases. Only then

does the whole setup of bringing in incidents within the development teams' workload make sense. Most organizations' release policy states that the changes to software can be made only during minor and major releases. Emergency changes are usually frowned upon and are viewed as exceptions in most organizations practicing ITIL service management. However, in DevOps-led projects, there are no exceptions; instead, there's only common sense. If a service or part of a service is down, then we need to bring it up at the earliest. For this to happen, if we need to deploy a package, so be it. What's the fuss over it? Amazon would ask. We do it multiple times a minute!

Following deployment, there are some sanity checks performed, some automatically and some manual. These checks are mandatory to ensure that the basic and critical parts of the system are working as they should. Nobody likes to make changes, even the Amazons and Netflixes. When a new package hits production, smoke tests are done to ensure that the scent of production did not drive the binary on a lunatic path (just saying)!

When the sanity checks wave the green flag, the incident manager informs the service desk to take the incident to closure by seeking confirmation from the user who logged the incident. If there is no user involved in this incident creation, then the appropriate customer contact is made aware of the resolution.

Through steps 5 to 10, the incident manager is actively across the incident. The person provides support and seeks updates from team members on a regular basis. The incident manager also keeps the customer updated on the progress and the expected date of resolution at all time. This communication with the customer is extremely critical as a loss of service generally results in revenue losses, and no business takes loss of revenue kindly. In my experience as a major incident manager, I have come across customers who have lost their cool, and the situation is far from pretty.

Step 11: Post-Mortem

The whole incident management process would be undone if there were no checks in place to identify the reason for the incident. However, this investigation is not done at a detailed level as the problem management process is tasked with investigations around service breakages. However, the incident management process conducts a fairly quick, though not thorough, investigation into the incident to find out the reasons behind the trigger for code changes. This is an important step in the DevOps methodology as the learnings from the investigation could forewarn the team before the next incident could possibly happen, or the investigation could lead to potential weaknesses in the code

and the logic. However, what is not done during a post-mortem is to point fingers at individual members of the DevOps team, pinpointing them for the lack of quality code and other mishaps that might have been the case.

The investigation is done during the sprint, and in the sprint retrospective session, the entire incident management process is put under the microscope including the outcome of the investigation.

What went right or wrong with the acceptance of the incident? Was it the right call, or should the incident have been sent back to L2 support?

Was the Scrum master and product owner informed quickly enough to allow maximum wiggle room in the sprint? Are there any improvement opportunities?

Was the incident well understood before beginning the development on it?

Were the testers involved when the incident details were briefed by the incident manager? Were the test scripts ready on time? Did the test scripts cover all scenarios?

These are some of the difficult questions that get addressed during the sprint retrospective sessions. Note that the questions are around the process and the outcome rather than around individuals.

CHAPTER 8

Problem Management Adaption

Incident management is the first line of defense in providing immediate relief against the disruption of services and eventual downtimes. However, by no means does the incident management process get into the nitty-gritty of putting an end to the cause behind the incidents. Its purpose is to bring the services back up, even if the solution is a nonpermanent one.

The second ring of process governance ensuring permanence to solutions is the problem management process. This is a process that deep dives into the cause of incidents and follows the problem to its root, ensuring that incidents related to the particular cause do not repeat.

To summarize, the incident management process deals with correction, while the problem management process focuses on prevention.

In this chapter, I will provide a brief introduction to the problem management process, the techniques involved, and the typical process, and then I will move into how the problem management process can be transformed in a DevOps project.

Introduction to ITIL Problem Management

The problem management process is featured in the ITIL service operations publication and is one of the critical processes for services to thrive. Incident management is good, but at the end of the day, the more incidents, the more downtime, and the more efforts required to bring the services back up. The customer gains zilch from the process as it's trying to keep the support above water at all times but not really taking it to new places. There is a dire need to bring value to services, and value can be brought about only if stability is assured. One of the pillars for ensuring service stability is the problem management process.

The problem management process is slightly academic. It does not believe in happenstance and trying to resolve world hunger in one go. There is a definite method to the madness around laying tombstones on top of the problems. I represent the problem management process as the investigation unit of the IT service provider organization. You might have seen the popular TV series *CSI*, where crimes are solved by following the leads and taking down the culprits. The problem management process is the *CSI* of IT service management, and you can compare incident management to the police squad.

Let's consider an example involving an application that crashes frequently when certain actions are performed simultaneously. An incident is raised. The incident is diagnosed, and the resolution is identified to be a complex one. But incident management focuses on helping users move on with their day-to-day activities involving services. Therefore, an incident analyst recommends a workaround in that the user can perform actions on the application in a sequential manner rather than in parallel. The user's immediate issue is solved, but the impending problem exists. A problem is raised, and an investigation into the problem begins. The problem is re-created, the codebase is examined, and all the relevant logs are studied to debug the underlying cause. The investigation pays off, and the cause is identified and subsequently fixed. All the investigative activities are done under the auspices of problem management to identify the problem and find a permanent solution.

Objectives and Principles

Before we get any further, the accurate meaning of the problem must be understood. The ITIL service operations publication defines it as the underlying cause of one or more incidents. In simple terms, there are incidents where the fix is yet to be found. The resolution of these incidents is not possible as the root cause of the incident is unknown. This is similar to a doctor prescribing medicines. If the doctor does not know what the cause of certain symptoms is, then the doctor will not be able to prescribe medicines. Likewise, to resolve incidents, the technical resolver groups must know the root cause of the problems. If they do not know the root cause, they start to guess by asking users to restart machines, uninstall and reinstall software, and other hara-kiri that may amount to waste of time and resources. But, if the principles of problem management were to be applied and the root cause identified, the solution to follow will be a matter of routine.

A problem gets raised when the root cause of an incident is unknown. Or a bunch of incidents with a common thread is unable to be resolved as the underlying root cause is yet to be identified.

Incidents vs. Problems

It is my experience that many IT professionals in the IT service management industry use the terms *incident* and *problem* interchangeably. This does more harm than good, especially if you are working in an organization that takes shape after ITIL and especially if you are preparing for the ITIL foundation exam. In this section, I will differentiate the two terms with examples, so as we move forward toward the process and other key terminologies, there shouldn't be any speck of a doubt between incidents and problems.

Incidents are raised due to loss or degradation of services. They are raised by users, IT staff, or event management tools. When the incident resolution is not possible, as the underlying root cause is unknown, the IT team will raise a problem. Remember that users and event management tools don't raise problems; generally speaking, they can come only through the incident. However, in a mature IT environment, we can configure event management tools to look for specific patterns of events and subsequently raise problems. But, let's keep this discussion out of the scope and restrict problems to be derived only on the back of incidents.

Let's consider the example of a software application that crashes when it is initiated. The user raises an incident to fix this issue. The software resolution team tries to start the application in safe mode, uninstalls and reinstalls the application, and finally make changes to the OS registry, to no avail. When all hopes fail, they provide a heads-up to the problem management process to find the root cause and provide a permanent solution.

The problem management process aided by experts in the software architecture group debug the application loading and run a series of tests to find the triggers and sparks for the crash. They find out that the root cause of the crash is because of a conflict with a hardware device driver. They recommend a solution to uninstall the hardware device driver and update it with the latest driver. The recommendation works like a charm, and the software application that used to crash loads nicely without any fuss. This is the problem management process in action, working on iron-legged problems that can cause irreparable damages to the customer organization if not dealt with on a timely basis.

Key Terminologies in Problem Management

Problem management digs deep, and the process brings a certain amount of complexity to the table. The complexity begins with it a few terms that are used quite often during various stages of the process activities. It is key that you understand all the terminology that I put forth here. It helps you use better terms at work and most definitely bags a few more right answers on the ITIL Foundation exam.

Root Cause

The *root cause* is the fundamental reason for the occurrence of an incident or a problem.

Let's say that you are in a bank and the ATM does not disburse the money that you requested. The underlying cause or the root cause for the denial of service in this instance is attributed to a network failure in the bank. Likewise, for every incident, there will be a root cause.

Only when you identify the root cause will you be able to resolve the incident. In the ATM instance, unless you know of the network failure, you cannot bring the ATM service back up.

Root-Cause Analysis

Identifying the root cause of an incident is no menial task. At times, the root cause may reveal itself, but many times, it will become challenging to identify the root causes of complex incidents. You are required to analyze the root cause by using techniques that commonly fall under the activity called *root-cause analysis* (RCA).

Remember that the outcome of RCA may not always result in the root cause of an incident. In such cases, RCA must be performed using complex techniques and with experts pertaining to related fields of technology and management.

Known Error

When the outcome of the RCA procedure yields results and the root cause is known, yet, it may not be always possible to implement a permanent solution. Instead, temporary fixes called *workarounds* are identified. Such cases where root causes are known along with the workarounds are called *known errors*.

There could be various reasons why solutions cannot be implemented. Commonly, permanent solutions come with an expensive price tag. Most organizations are price conscious these days and may not approve the excess expenditure. Other reasons could include a lack of experts or people resources to implement the permanent solution or could cite governance or legislation controls that could prevent implementation.

Known Error Database

Known errors are documented and are stored in a repository called a *known error database* (KEDB).

The KEDB consists of various known errors, their identified root causes, and the workarounds that can be applied. The known error records are not permanent members of a KEDB. Known errors will cease to exist in this repository when the permanent solution is implemented.

Workaround

As I mentioned, *workarounds* are fixes to solve incidents temporarily. Each incident could have one or multiple workarounds, but none of them will alleviate the problem permanently, and it may be required to revisit the workaround applied on a regular basis.

For example, say a printer on your floor is not working and you cannot wait until the technician can get to it. A classic workaround, in this case, is to print from a printer on a different floor. The workaround will solve your problem temporarily by providing a way out, but it may not be a permanent solution as you may find it inconvenient to run down to the next floor every time. Another workaround could be that you don't print the document but instead send the soft copy to the intended recipient.

Permanent Solution

When the root cause of a problem is known, the follow-up activity in problem management process is to identify a *permanent solution*. This solution permanently resolves the problem, contributes toward a reduction in the incident count, and avoids future outages.

As I mentioned earlier, permanent solutions come at a cost, and organizations may not always be willing to shell out the required capital. In such cases, permanent solutions are known but not implemented.

Problem Analysis Techniques

There are a number of ways to investigate a problem. Every problem is unique and may require a different approach altogether to determine the cause of it. My favorite investigative approach is to use common sense and follow the trail until it leads you to the smoking gun---in some ways like Sherlock Holmes conducts his investigation. I don't like to be boxed in with approaches that come attached with a model such as a theme for investigation; examples are the forensic anthropology used in *Bones* and the mentalism illustrated in the show *The Mentalist*. Yet, it is good to know about the approaches so you can understand the methodology and create your own techniques to solve a problem. In this section, I will introduce a few popular techniques used by investigators in the IT industry, most popularly known as *problem managers*.

Brainstorming

The technique that has been used, misused, and underused at times is the power of using our brains to focus on areas of investigation. The brainstorming technique involves focused thinking without any inhibitions. The term *brainstorming* was first made popular by the author Alex Faickney Osborn in the book *Applied Imagination*, published in 1953. Although the methodology around brainstorming existed long before, it was never brought into the limelight as a powerful technique to investigate problems.

In the brainstorming technique, there are no bad thoughts. Every single thought must be weighed, and then a decision must be taken. In other words, ideas are not tagged absurd or made fun of, and everything is accepted, examined, and then acted upon based on the results of the thought. Let me explain brainstorming in the form of an example. If thinking is a car, then in this car I take out the brakes because I don't want the thinking to stop, or be impeded. There must be no action taken to stop the flow of thoughts. I use only the steering wheel to steer my thoughts toward the goal I want to achieve. The more thinking, with the right steering, the closer I will get to my destination.

Brainstorming can be done on your own or in a group. The more the merrier, right? Not always. It is possible through group brainstorming sessions that the clear thoughts in your mind could get defocused, so group brainstorming must be done with caution and with a process to keep it in a framework. Osborn in his book says that group brainstorming sessions are more effective than individual ones as he firmly believes that quantity breeds quality. The assumption is that a greater number of ideas generated

provide a better probability of striking gold. If you are planning on a group brainstorming session, then I recommend following these steps:

1. You need the right set of people to brainstorm an idea or to investigate the cause of a problem. If you bring like-minded people into a brainstorming session, you are not going to get too many ideas to resolve problems. Instead, if you bring in a diverse group of people, thoughts from different prisms flow through like water, giving you all the ideas needed to investigate a problem.
2. After you firm up on the people, don't bring them together into a meeting room without preparation. Explain to them in advance what the goal of the brainstorming session is and what you want to achieve. This agenda setting will help the group to start thinking even before they set foot into the brainstorming session, and their ideas can be discussed, dissected, and challenged, which is a lot more productive than doing the actual thinking in the session.
3. In the brainstorming session, restate the agenda and the set the rules of engagement. Rules of engagement can be something like this: no ideas are bad, and none of it should be cast away without examining it. This is an important step to provide a voice to the participants who may sit on the bottom steps of the corporate ladder.
4. Use a whiteboard to note down every single idea. Note an idea, discuss it, and then mark up or mark down the idea based on its merit. I have used mindmaps (made popular by Tony Buzan) and Kanban dashboards on the whiteboard to help me visually organize and manage the ideas.
5. As a chairperson of the brainstorming session, you must have a good idea of what you are trying to achieve. You must use this knowledge to steer the group toward ideas that matter. Remember the example in which I talked about the steering wheel is the only control on the car of thoughts. You need to be this steering wheel. You need to guide the discussions. Appoint a note-taker to jot down all the ideas that have been prioritized higher for further discussion. Take as many breaks as needed. Our brains work better in short sprints rather than in marathon sessions.

Five-Why Technique

One of the most used and misused techniques in the problem management process is the five-why technique. It is used during the investigation of a problem, specifically during the root-cause determination stage. The technique is so commonly taught and retaught to problem management personnel that it has become a de facto standard in the activity of root-cause analysis.

The five-why technique involves asking the question “why?” to the problem on hand five times to arrive at the root cause. It was conceived by Japanese industrialist Sakichi Toyoda, the founder of Toyota Industries in 1930. But it wasn’t until the Toyota Production System became popular that the technique became popular in the 1970s. The technique’s principle relies on being on the ground to find out the reasons rather than in the comfort of an air conditioner (a “go and see” philosophy).

The part that makes the technique popular is that it is extremely simple to use and takes a short amount of time to process and execute it.

Applying the Five-Why Technique

Let’s consider the most common problem in the airline industry. Most airlines have the problem of dealing with flight delays. Flight delays not only inconvenience passengers but also tarnish the image of flight operators, which works against the optimal use of their assets and leads to disrupting plans of their expansion. In this example, I have considered a simplistic view of the problem and have arrived at the cause of the problem using the five-why technique, as shown in Figure 8-1.



Figure 8-1. Five-why technique for determining the root cause of flight delays

My first step and perhaps the most significant activity is to identify the problem accurately. The problem must be pinned down to its granular details if possible to have a better chance of identifying the root cause. The problem chosen in the example is rather simplistic and at a high level: low percentage of on-time flight arrivals for a particular flight operator. Perhaps going by the metrics, I could have been a lot more specific with the problem statement: 17 percent on-time flight arrivals for flights departing from the Kempegowda International Airport. By being specific, I get the opportunity to put on a microscope, get on the ground, and examine the reasons for the problem.

In this technique, I question the reason for the problem's existence. Why are flights departing the Kempegowda International Airport later than the planned time? The immediate answer throws light on the baggage check-in process that seems to hold up passengers before arriving at the gate. So, it becomes a moral responsibility of the airline to wait for those passengers in the queue at the baggage check-in. Thinking through it, the next logical question to ask is why are people spending so much time in a queue? Is this normal for other airlines as well? The answer to this question points toward limited counter space that the airline has set up compared to its competitors, which are operating a similar number of flights. The next obvious question we want to understand

is why the airline operator chose fewer counters while it fully knew that other airlines had a significant number of check-in counters. The answer to this question is rather painful in this example. The airline is facing financial difficulties and, as a result, has made a number of cutbacks including cutting down on the baggage check-in counters. *Voila!* We have the cause of the problem. Or do we? Why am I not digging in deeper to find out why the airline has financial problems? Maybe the root cause could be financial mismanagement. I can dig deeper, but in this example, I choose not to. The five-why technique proposes that I question the problem with “why?” five times, and more often than, not even before I ask the fifth “why?” I should have the root cause. In some cases, I may need to ask the question “why?” a few more times to pinpoint the cause. The reason I am choosing to not dig deeper into this flight arrival problem is that the cause of the budget is good for me to work with. Going into the financial details may not give me an immediate remedy to the problem I am facing. To summarize on the number of whys, use the number wisely and do not stick to five just because the technique highlights a particular number. This is not a prescription but rather guidance toward solving a problem.

Based on the cause that is identified (the low budget), an appropriate solution can be conceived. It could be online check-ins with baggage drop-off stations, self-serve baggage check-in counters, or shared resources with other airlines to have common check-in counters for all airlines. The effectiveness of the solution solely depends on the quality of the root cause identified. Therefore, it is critical that the root cause identified is as specific enough to offer a full-blown real solution that isn’t superficial and molded for the sake of a solution.

Limitations of Five-Why Technique

While the five-why technique is popular as the model is easy to adapt and simple to use, it begs a question whether this technique can be used to solve problems of higher complexity. Of course, we can further expand on the technique in the flight delay problem if the “why?” question gives rise to multiple reasons. We can use multitiered “why?” questions to determine the cause of the problem.

However, the real test of the technique is when the person answering the question does not know the answer to the question. The person might not even know where to look. The technique just asks the question “why?” but does not supplement the question with helpful keys to help determine the cause. For example, the answer to flight delays may perhaps be the efficiency of people manning the check-in desk. Did the technique

attempt to find out whether the reasons could perhaps be any other than those given by the person trying to solve the problem? This is why I feel that the five-why technique is limited when there is a complex problem with multiple tentacles. With this, I want to introduce the next problem-solving technique: the Ishikawa diagram.

Ishikawa

An Ishikawa diagram is known by multiple names such as fishbone diagram, fishikawa diagram, and herringbone diagram, among others. The diagram consists of a central spine that represents the problem. Several branches jut out of the spine to indicate possible causes. The arrangement of the spine and branches looks like a fishbone (see Figure 8-2).

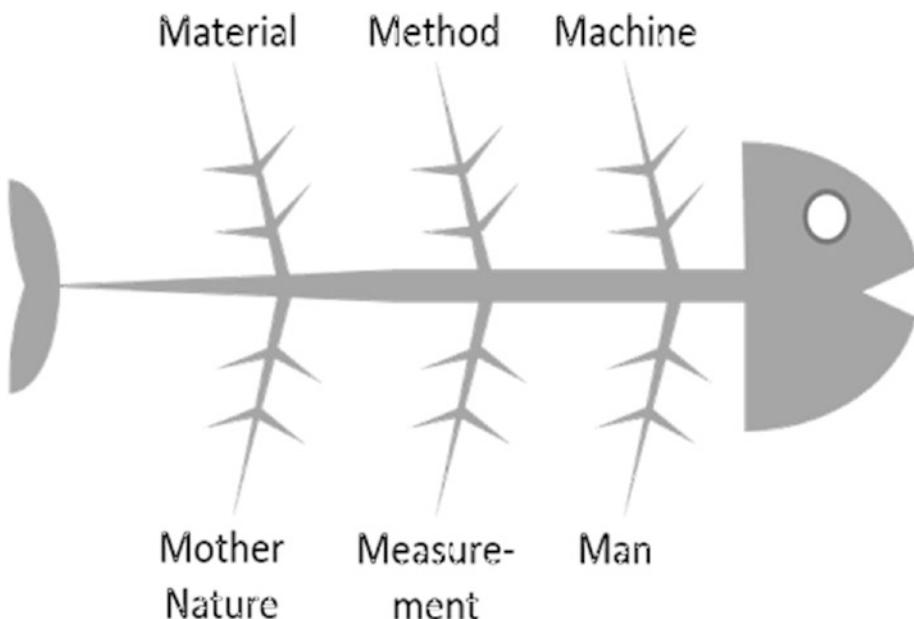


Figure 8-2. Fishbone diagram (image credit: 4improvement.one)

The causes are not arbitrary as discussed in the five-why technique. There is a method to the madness in the Ishikawa process of the root cause of determination. Each branch is designated to a category of cause, and the thinking behind it is to follow the category lead to determine the root cause. Figure 8-2 illustrates one of the popular

CHAPTER 8 PROBLEM MANAGEMENT ADAPTION

fishbone models used in the manufacturing industry called the 6M model. The six categories of causes that are modeled are as follows:

- Material, indicating causes related to the material used in the manufacturing process
- Method, the process
- Machine, the actual machinery, technology, and so on
- Mother Nature, the environment
- Measurement, the measurement techniques employed in deriving metrics
- Man, the people involved

There are other models as well depending on the type of industry. For example, the service industry uses the 4S model with these categories:

- Surroundings
- Suppliers
- Systems
- Skills

The service and marketing industry is also known to use the 8P method with these categories:

- Procedures
- Policies
- Place
- Product
- People
- Processes
- Price
- Promotion

I am generally comfortable with the 6M process, and Figure 8-3 will use this model to explain the Ishikawa diagram. However, do not feel restricted to use one of the available models. Come up with your own set of categories to help you assess the root cause of the problem. These models are to be used as a starting point, and this should eventually give way to your own set of categories for the industry, domain, and customers you are involved with.

Using the same example as earlier (the low on-time flight arrival rate) and using the Ishikawa diagram with the 6M categories, Figure 8-3 depicts the possible outcome of the analysis.

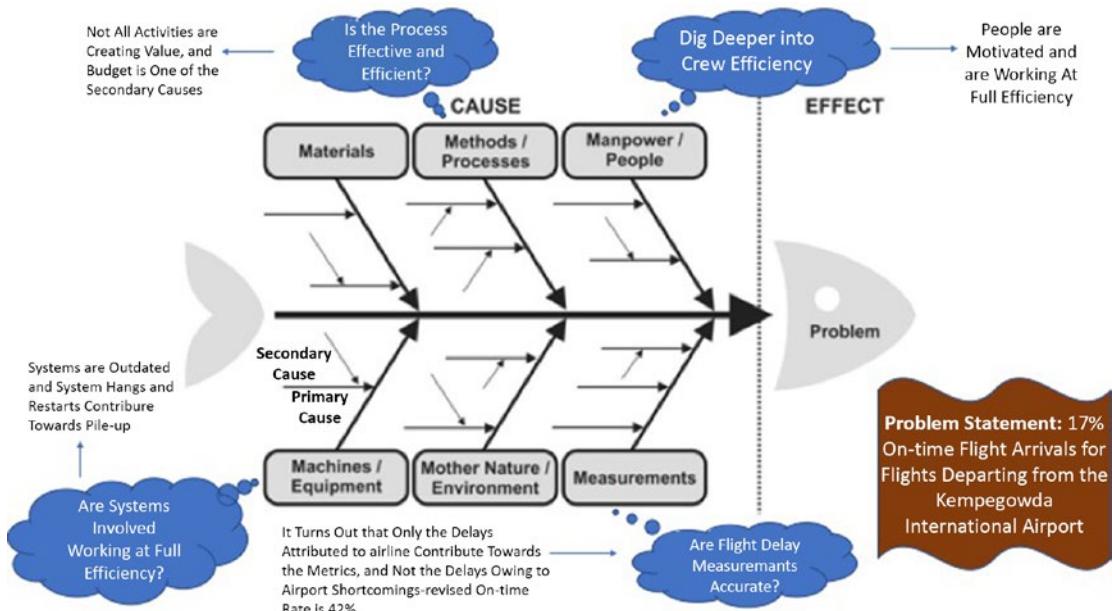


Figure 8-3. Ishikawa analysis for low on-time flight arrival rate

In this example, the problem is central to the whole exercise. Under each of the branches, I look at the problem from a different perspective. For this particular example, I find manpower, methods, machines, and measurements to be relevant. Therefore, my analysis surrounds these categories only.

Note Every problem is unique, so you need to carefully pick and choose the branches for the analysis of the root cause. Don't find it a burden to explore every single category. Be judicious and use common sense in identifying the right categories.

When we consider a perspective to view the problem, say manpower, we start asking a number of pointed questions to arrive at a root cause.

- Are people sufficiently trained to do their jobs?
- Are people motivated on the job, and are they compensated fairly?
- Are there sufficient people to man the check-in counters?

The answers to these questions will take me closer to finding the root cause.

Remember that there is no set formula for probing a problem. You must use logic and work on the ground to understand how things work.

My analysis into manpower tells me that people were sufficiently motivated, trained, and quantified, so manpower is probably not the reason behind low on-time arrivals.

I move onto the next category, which includes the methods or the processes involved. My analysis into the processes side of things reveals that there were certain activities that the check-in personnel were doing that was not yielding any value or meeting the objectives of the process. They were dead activities that just consumed time and effort. Probably optimization to this effect could help with the efficiency. Another revelation, of course, was the fewer counters and the budgetary problems. These two actions look strong at this point in time.

Under machines, a careful examination and analysis of the infrastructure and application environment revealed that legacy systems were employed, and processing passengers was not a smooth process after all. Plus, frequent freezing of systems and restarts did not help reduce the check-in queue.

Finally, under measurements, it was found that the percentage of 17 percent was indeed faulty. It did not exclude the delays attributed to the airport. By excluding them, the problem statement changes slightly for the better, with 42 percent on-time arrivals for flights departing from the airport at Bangalore.

Now that I have the causes handy, I can get to work identifying necessary solutions to the revelations. There exists a single root cause to a problem. However, for problems such as the one considered in this example, it is possible that multiple causes are adding to the delays, and tweaking various elements does help improve the on-time arrival rate.

Kepner-Tregoe

The Kepner-Tregoe method is yet another popular technique that is in vogue to analyze problems and make appropriate decisions. It is a problem-solving and decision-making (PSDM) technique that decouples the problem from the decision taken to deal with it. This technique was developed by Charles Kepner and Benjamin Tregoe in the 1960s.

There are four steps in the Kepner-Tregoe method to make the best possible decision.

1. *Situation appraisal:* In the first step, the problem situation on the ground is analyzed. What exactly led to the current situation? At this point in time, we do not talk about the problem itself, but the outlining of concerns is brought forth here.
2. *Problem analysis:* In the second the step, problem analysis, we find out the root cause of the problem. The problem is analyzed, and the root cause is determined, possibly using one of the methods that I discussed earlier.
3. *Decision analysis:* Based on the root cause identified, various alternatives for resolving the problem are identified. In an unbiased manner, each alternative is weighed to calculate the risks and increase the benefits.
4. *Potential problem analysis:* Against the best alternatives that are identified in the previous step, further analysis is performed to identify whether there are going to be any regression issues based on the dependencies that exist.

At the end of the four steps, a decision is made to go with the best possible alternative that was identified in the process. This is a process that tries to be unbiased in terms of identifying the approaches for resolving problems and looking at the resolutions objectively. However, some critics rightly say that the decision-makers are always inclined to one side or the other, making it a biased approach after all.

However, the method helps put out options on the table, along with the associated risks. The wise decision-makers will have all the information they need to make the right decision.

Typical Problem Management Process

The problem management process can be drafted in a number of ways to meet the goals of the problem management process. While ITIL stays away from prescribing how the flow must look, it provides a true north for the process architects to follow. A typical problem management process workflow with all the bells and whistles is illustrated in Figure 8-4.

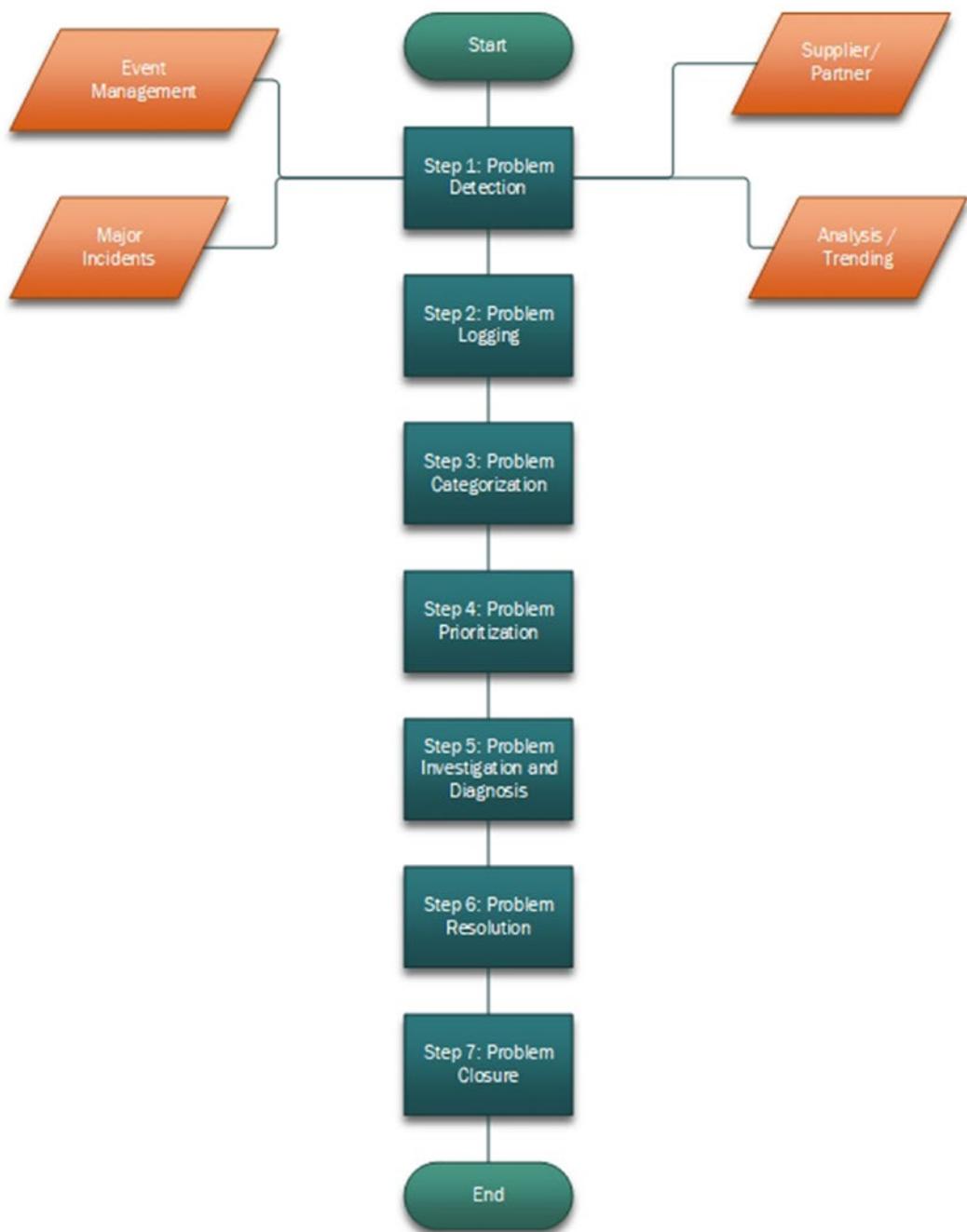


Figure 8-4. Typical problem management process

Step 1: Problem Detection

As with the incident management process, problems need to be detected for the process to be triggered. A problem can be identified from any source; it can come as an action item from the customer, a threat notification from regulatory departments, or loopholes identified by hackers. In the problem management lifecycle, I have considered the most common triggers for a problem. Remember that the triggers need to be identified beforehand to ensure that the process is well controlled and does not spiral out into directions that were not factored in.

Event Management

These days, event management tools play a handy role in keeping a finger on the pulse and to standardize monitoring and capture events that are of significance to IT service management. We have seen the application of event management tools in the incident management process. In a similar vein, these tools can be programmed to detect problems as well. For example, if a server being monitored goes down, an incident ticket is raised. Suppose the same server alternates between being offline and online a preconceived number of times; the tools can be programmed to raise a problem ticket for the problem manager to start investigation-related activities.

With all this said, it is rare to see the event management tools used to raise problem tickets. I have seen such instances with a handful of implementations. Programming problems on automation solutions requires a certain amount of maturity for the service management organization to design, execute, and control.

Major Incidents

The most common source for triggering problems are major incidents. It is a common sight in the process world to see major incidents tagged to problem investigations. Normally, toward the resolution of a major incident, a problem ticket gets created by the incident manager. While the major incident resolution brings in a logical closure to the major incident management process, it gives rise to the problem management process to kick-start and initiate investigations.

This is a good practice as major incidents can cause massive damage to clients (financial, productivity, brand image, and so on). It is in the interest of all stakeholders that a problem investigation is performed on the major incident to identify the loopholes and come up with a preventive measure to ensure such outages do not happen in the future.

Partners/Suppliers

We live in a world that no single IT service provider organization can afford to provide all IT services based on the supplier's expertise.

Suppliers handle specific areas in the delivery of IT services. Let's say that a particular supplier provides networks, a supplier provides infrastructure, and a supplier manages applications. These suppliers are the best sources for identifying problems in their respective areas, rather than a third party identifying it. Suppliers are one of the main triggers or sources of detecting problems. Suppliers are also referred to as *partners* to make them accountable for the overall delivery of IT services.

Analysis/Trending

Proactive problem management's objective is to forecast future problems and stop them from happening. In the movie *Minority Report*, the aim of the main crime-fighting organization is to stop crimes before they actually take place. This concept of problem management is similar to that movie theme.

How do you forecast what is going to happen in the future? We don't have crystal balls in IT, nor do we have precogs as in the movie. But we possess historical data. This data can be dissected, and when cross sections are analyzed, we can see into the future.

One of the techniques for doing proactive problem management is to trend the incidents or the common root causes of incidents. This will provide us with insight on what is generally going wrong. If we can concentrate our cognitive powers on these frequent occurrences and devise a way to fix the problem permanently, then we have reduced the recurring incidents. Along with it, we have reduced the incident count, potential outages, potential penalties, and potential brand image scarring.

Another technique is the Pareto principle, where we identify the top 20 percent of the causes, which normally are responsible for 80 percent of the incidents, and find a permanent solution for the identified incidents. If you could do this, you would reduce the incident count by 80 percent.

Step 2: Problem Logging

Problems that are detected need to be documented in a formal way to ensure that each problem goes through all the steps in its lifecycle.

Every problem ticket is likely to have all or a subset of the following attributes:

- Problem Number (unique)
- User details
- Problem Summary
- Problem Description
- Problem Category
- Problem Priority
- Incident Reference Number(s)
- Investigation Details
- Resolution/Recovery Details

Event Management

Most event management tools today have the capability to auto-log problems on the ITSM tool. If this capability is not available, the tool will raise alerts in the form of e-mails to the service desk function.

For major incidents, problems are logged by the incident manager or the service desk function, generally toward the end of the incident resolution

Partners/Suppliers

Suppliers or partners who identify problems either log problem tickets on their own or provide the necessary inputs to the service desk function or the problem manager.

Analysis/Trending

Problem managers who perform the analysis activities raise the problem ticket based on their findings.

Step 3: Problem Categorization

All problems have to be categorized similar to incidents. Categorization will help in assigning the problem tickets to the right resolver groups and in reporting.

Step 4: Problem Prioritization

Some problems are more important than the others. They need to work with more focus on the others. How do we differentiate one problem from another? Through assigned priorities. This exercise is similar to incidents.

Similar to the incident priority matrix, a problem priority matrix exists, and a timeline is associated with it to set targets for investigation and resolution. However, in the service industry, problem timelines are not strictly adhered to like with incidents. This is because investigation tends to take longer than expected in the case of complex problems, and generally, there will not be enough resources assigned to problem management specifically but rather shared with incidents and changes. When an incident comes up during the same time as a problem, the incident always takes priority, and resources will always end up falling short on the time needed to investigate problems and come up with a permanent solution.

Step 5: Problem Investigation and Diagnosis

The step of problem investigation and diagnosis starts with identifying the root cause of the problem. Getting to the root cause of the problem is the biggest challenge in this exercise. A root-cause analysis (RCA) is the output of this step, and it involves various RCA techniques that are employed in getting to the root cause. Five-why analysis, Ishikawa diagram, Pareto analysis, affinity mapping, and hypothesis testing are the popular techniques used. The discussion of the techniques is outside the scope of the ITIL foundation examination and hence this book.

To conduct a thorough investigation into a problem, suitable resources who are referred to as *problem analysts* must be assigned. They are technical experts who have the expertise to delve deeper into the cause of the problem. They will be aided by the knowledge management database (KMDB) and the configuration management system (CMS). The KEDB will also be referred to when identifying whether similar problems have occurred in the past and what resolution steps were undertaken.

In most cases, the problem analyst tries to re-create the problem to identify the root cause. After identifying the root cause, problem resolutions are developed, preferentially economic solutions.

Step 6: Problem Resolution

In the previous step, the root cause of the problem is identified, and a solution is developed to mitigate the problem. The solution can be either a permanent solution, which is preferable, or a workaround. You should also be aware that the solution implementation may come at a cost. The client might have to invest some capital into resolving problems, maybe adding extra infrastructure, procuring applications, developing connectors between applications, and leasing more bandwidth, among others. So, a financial approval from the sponsors will always precede the problem resolution activity. The financial approval will be based on the business case that the service provider develops and the return on investment that the resolution brings to the table. For example, if a particular resolution costs a million dollars and can improve the client productivity by 10 percent, the client might be tempted to approve it. On the other hand, for the same million dollars, if the return on investment cannot be quantified, it may not get the nod.

Implementation activities will be carried out through the change management process. The resolution will be submitted to the change manager in the form of a request for change (RFC). The change management process will conduct due diligence like risk and impact analysis to ensure that the resolution does not cause more harm than good and that it does not impact other connected services (regression analysis) or cause outages during business-critical periods. Most solution implementations go into the change advisory board (CAB) approval cycle for further assessments and scrutiny. After getting the okay from all stakeholders, it will be taken up for implementation.

Resolution can also come in the form of a workaround. I discussed workarounds earlier in this chapter. Workarounds too will face the music from the change management process.

At the end of the resolution activity, the KEDB will get updated. If a permanent solution is implemented, the KEDB record will be archived. If a workaround is implemented, the KEDB record will be updated with the necessary workaround details. In a workaround implementation, it is a good practice to keep the problem record open.

Step 7: Problem Closure

When a permanent solution is implemented, the problem record needs to be updated with the historical data (of the problem), resolution details, and change details; then it is closed with the appropriate status. If a workaround is implemented, keep the problem

record open in an appropriate status to indicate that the problem is temporarily fixed using a workaround. The problem manager is generally responsible for closing all problem records.

Problem Management in DevOps

The problem management process exists to reduce incidents and to ensure that the IT environment is free from anomalies that hold back the system and the services. The process is relevant in an IT services environment powered by ITIL, and a DevOps project is an extension of the ITIL environment. So, it is clear that problem management is here to stay in a DevOps project.

You may think I am contradicting myself based on what I said earlier about DevOps. I mentioned earlier that the goal of DevOps is to deliver fast and to minimize the number of bugs in the production environment. It is impossible for any methodology to claim that a bug-free system is going to be deployed. If that's the case, then there is no room for either the incident or problem management processes. But, bugs are an inherent part of the system. They are like the weeds in your garden. No matter how technologically savvy you are in agriculture, you can never take all the weeds out. The same is true for the bugs in an IT system. They are here to stay and so is problem management to minimize the effects of these bugs.

What Are the Possible Problems in a DevOps Project?

Incidents and problems often have the same hue as a DevOps project. The speed of delivery is the most important aspect. So, when incidents flow in, they come in quickly, and they move out quickly as well. The time that is needed to analyze the incident is kept short to avoid extended downtimes, and this often gives rise to a dilemma of not sharpening the axe before felling the tree.

Of course, there are bugs (and incidents) that crop up for which the developers may not have the answers to. In the interest of moving ahead and reducing the backlog, the developers might put the bug on the backburner, tagging it as an "unresolvable." These unresolvable defects form the basis for reactive problem management. As the word *reactive* suggests, you react to the problem on hand rather than anticipating it and removing the cause of it before the problem materializes.

CHAPTER 8 PROBLEM MANAGEMENT ADAPTION

The enigma over incidents and problems needs to be reexamined in a DevOps project. In an IT services project, the boundaries are straightforward. Problems are incidents that cannot be resolved permanently or resolved at all without detailed analysis. A post-mortem on major incidents is done on the back of a problem.

Analysis on the incidents can reveal certain interesting aspects such as incidents that have commonalities, often called *repetitive incidents* in ITIL lingo.

In a DevOps project, what can be categorized as problems?

Incidents flow from the L2 support to a DevOps team. The DevOps team works on it for a reasonable time based on the incident priority. The planning around working on incidents is taken up the Scrum master and the incident manager. When the developer spends sufficient time on an incident without success, the Scrum master, in consultation with the incident manager, might weigh in on the impact of the incident and workaround if available. Instead of spending excessive time on an incident that does not have a major impact, the Scrum master will make a decision to rebadge the incident as a problem, and the problem goes back into the backlog, only to be picked up in one of the future sprints. This is one source of problems.

Major incidents are rare in most projects. However, when one does happen, there needs a good amount of post-resolution analysis to ensure that major incidents are avoided at all costs. Post-resolution analysis is a necessary evil, and the problem management process is best equipped to deal with it. This is equally applicable to DevOps projects as well.

I started this topic with an axe-sharpening analogy. It is so true that we need to sit back and take a holistic view of the flow of incidents. Only then we will get the true effects of axe sharpening and don't slog away at incidents instead of bringing an end to a bunch of incidents in one swooping move. The analysis of repetitive incidents that I mentioned earlier is a worthy exercise in a DevOps project, and it helps with the stability of the system and helps deal with the problem head-on rather than in a reactive manner, which makes a case for a dedicated role to manage problems.

The Case for a Problem Manager

There are discussions on various forums about the relevance of full-blown ITIL in a DevOps world. One of the liabilities according to a certain school of thought is the role of a problem manager. The detractors say that the problem management process is relevant and has value, but having a dedicated role of a problem manager to manage the problems is overkill because the problem manager does not do the heavy lifting that is involved in the analysis of problems and root cause identification. The person merely touches the surface and acts as a record keeper. Well, in my humble opinion, these detractors are wrong. There is considerable value in having a problem manager, and in this section, I hope to prove them wrong.

I spoke briefly on reactive problem management in the previous section. The word *reactive* must not be looked at as something to stay away from. There is value in reacting to problems on hand and ensuring that they are resolved quickly. Reactive problem management takes place on two fronts in a DevOps project.

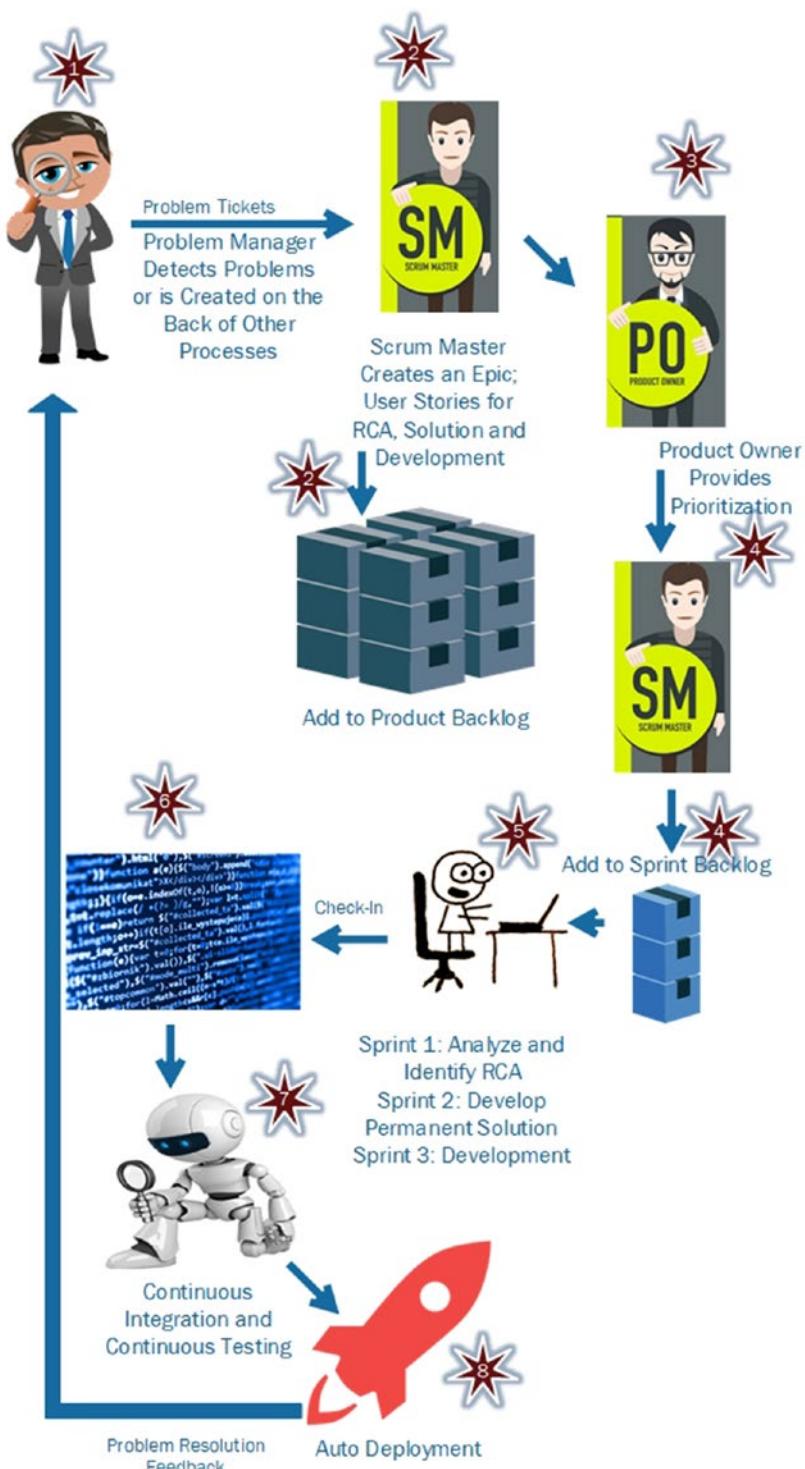
- Defects that cannot be resolved (at least quickly), wherein the impact of the bug is minimal (such as slight misalignment of the footer text). However, these defects must be fixed at some time or the other, sooner than later. Tagging such defects by an appropriate status such as “unresolvable” will provide an opportunity for the planning team (Scrum master) to break down the problem into multiple parts and present them in various sprints for action and closure.
- I presented the case of major incidents in a services project as being appended with a problem ticket to conduct investigations and propose a permanent solution to ensure such major incidents does not repeat. This scope of problem management can be extended to a DevOps project as well. Not only major incidents, there’s more coming into the realm of problem management from a DevOps project. DevOps projects are generally anal about defects getting into production. Incidents that are raised against software releases need to be looked at under a microscope. Who best to do it than problem management?

Proactive problem management in DevOps is about the analysis of incidents. Incidents are analyzed based on various common factors such as the nature of the incidents, the configuration item involved, the application features, and so on. This sort of analysis more often than not provides an opportunity to get an overview of incidents. The results are always interesting because you start to find problems in areas where you least expect. As the Pareto principle goes (tailored to ITIL), 80 percent of the problems are caused by 20 percent of the incidents. So, it is important to identify that bugger behind a whole bunch of incident!

Problems do not make a lot of hue and cry as the incidents do. Therefore, it is anything but natural to do away with a problem manager as there isn't much noise around problems. This is how the IT industry operates. The ones who make the maximum noise get the prize. The problem manager is like a campaign manager, working in the background and drumming up support for your candidate. Thus far, I have established the role of problem management in the overall scheme of things, and looking at the activities surrounding reactive and proactive problem management processes in a DevOps project, a dedicated problem manager to shepherd the problems to its closure seems to be the best logical choice. However, it is also true that with the advancement of DevOps methodologies and catching defects early, the defect rate has dropped, and a problem manager being a part of DevOps teams seems rather a luxury than a necessity. In Chapter 5, I proposed multiple layers of team structures. The DevOps team is dedicated to a single product and shared teams managing multiple DevOps teams. The problem manager can be part of the shared teams and manage problems from multiple DevOps projects if there isn't sufficient load from a single DevOps project.

The DevOps Problem Management Process

The problem management process in a DevOps project is in principle the same as in an ITIL-driven services project. The process, however, gets broken down into multiple sequential items that are processed in sprints based on their prioritization. A rundown of a problem management process in a DevOps project is illustrated in Figure 8-5.

**Figure 8-5.** DevOps problem management process

Step 1: Problem Detection

Incidents are identified while problems are detected. Problems are not obvious; they need to be unearthed based on historic trends or on the back of patterns. The problem detection is primarily entrusted to the problem manager who is part of the shared team. However, detecting problems is the responsibility of all involved personnel in the delivery of the project. Note my words here. I am not just saying developers or testers but *everybody*. Problems can occur anywhere, be it in the toolsets or infrastructure or the way the Agile implementation is done. After it has been detected, the problem manager takes ownership of the problem and analyzes it for its relevance, value, and impact. If it is low on all three counts, then the problem manager may close down the problems without further action. For problems that rank high as per the pre-agreed matrix, the problem manager decides to take them further.

One of the most common ways of identifying problems is through trend analysis. A trend analysis for the past six to twelve months will give out patterns that can be used to identify the common denominator. A problem ticket is raised against the detected problem. This type of problem management where problems are detected (and rectified) before they bloom into major impacting incidents is called *proactive problem management*. The problems detected and solved on the back of a major incident (or coming from other processes) is referred to as *reactive problem management*.

Step 2: Scrum Master Logs the Problem into the Product Backlog

The Scrum master receives the problem ticket and for every problem ticket creates an *epic* (a big chunk of work consisting of multiple user stories) on the product backlog with the following user stories:

- Analysis (RCA)
- Solution
- Development

I recommend creating a minimum of the recommended three user stories because a problem needs analyzing to identify the root cause. Identifying the root cause is potentially time-consuming and can eat away a developer's time for the entire duration of a sprint. So, while planning, the Scrum team can sequentially take up analyzing the root cause in sprint 1, developing a solution based on the identified root cause in sprint 2, and developing the solution and testing it in sprint 3. The order cannot change, but the

sprint numbers can. Based on the prioritization, the Scrum team can pick up root-cause analysis in sprint 2, the solution in sprint 5, and its development and testing in sprint 7. The order cannot be turned around; in other words, the solution cannot be developed before the root cause is identified.

Steps 3 and 4: Product Owner Prioritizes the Problem and Is Added to Sprint Backlog

No surprises here. The product owner is the sole proprietor of the product backlog and prioritizes epics and user stories. In this case, the product owner has decided to prioritize the problem over all other product backlog items. The user story pertaining to analyzing the root cause (followed by other user stories in subsequent sprints) is added to the sprint backlog.

Step 5: Scrum Team Acts on the Problem

It is okay that the Scrum team instead of developing and testing user stories puts on a Sherlock Holmes hat and gets into the investigation of the problem. The definition of done in this case will be to identify the smoking gun.

When the root cause is identified, the next user story involving developing a permanent solution for the problem based on the root cause is prioritized and added to the sprint backlog. The development of a permanent solution can be done by either developers, testers, operations, or architects (who sit in the shared teams). During the sprint planning session, the person who is going to derive a solution is identified, and the definition of done is to come up with a solution that plugs the problem.

In the subsequent sprint, the solution derived is developed and tested if it involves coding. If it involves infrastructure changes, then the respective team member gets down to work to implement the recommended solutions.

Steps 6, 7, and 8: Continuous Integration, Testing, and Auto-Deployment

If the solution involves coding changes, the changes made to the mainline are fed back into the mainline, the continuous integration and continuous testing processes take over, and the binaries are deployed into environments of choice.

CHAPTER 8 PROBLEM MANAGEMENT ADAPTION

The problem manager is kept in the loop throughout the process. Although this problem manager will have a limited role to play in the process, this person will keep monitoring to ensure that the process and deliverables are as per the agreed-on quality and timelines. When the solution is successfully developed and implemented, the problem manager closes the problem ticket as successfully implemented.

CHAPTER 9

Managing Changes in a DevOps Project

They say that change is the only constant. They also say that anything that does not grow withers away. This is so true in the software industry. No matter how old the software or the service is, changes to it happen all the time. No matter how legacy the application is, it still needs to be maintained and made current to the organizational needs.

Changes are inevitable in any industry; therefore, the onus is on management; the question is not whether we make changes or not but rather how do we do it without impacting the product or the service negatively.

It is also true that a majority of incidents are caused by mismanagement of changes. So, that is all the more reason that we need to tighten up the change management process to increase the overall uptime and reduce the number of outages. In this chapter, I will provide the necessary details of the ITIL change management process, including the types that exist and a typical process that is normally employed. Then I move into adapting the change management process into a DevOps project.

To me, a real difference can be made in the way we deliver projects, and the change management process is a major stakeholder in ensuring that the security blanket that it provides protects the service from malicious changes. This is the process that sets the bar for implementing DevOps successfully in organizations; it does this by allowing seamless changes with the least amount of bureaucracy and by providing plenty of support to guard against mishaps. In the ITIL adaption scheme for processes, the change management process will define the way forward.

What Constitutes a Change?

A change can come in many forms. Identifying a change as a change is half the problem solved because most organizations fail to define the list of changes as compared to other forms such as service requests. So, the big-ticket question is, what is a change?

Overview of Resources and Capabilities

Changes in the ITIL framework and changes in the project management framework have different connotations. My focus in this section is ITIL; therefore, the change that I refer to here is the change that comes about for a service. A service is made up of multiple moving parts, and in ITIL they are broadly classified as resources and capabilities.

Figure 9-1 lists resources and capabilities, and together each of the items listed under them is called a *service asset*.

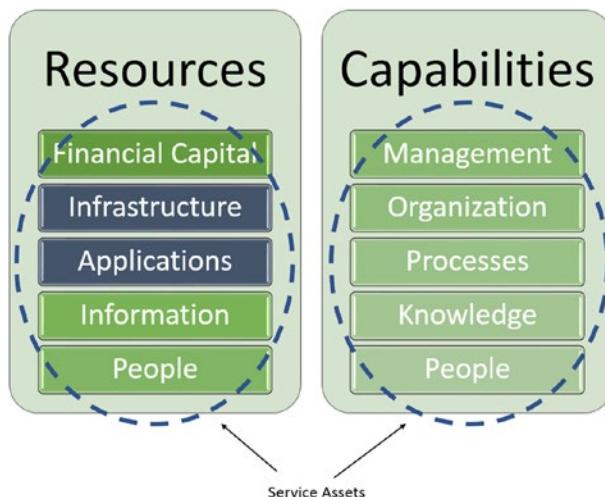


Figure 9-1. Resources and capabilities of a service

Note People are listed under both resources and capabilities as they directly make up the resources to support and build services, and also their capabilities have a major stake in the quality of their delivery.

The ten service assets listed in Figure 9-1 represent the various moving parts of a service. Changes to any of the ten service assets are likely to have an impact on the service. While some service assets impact services directly, most impact it indirectly. For example, financial capital is a necessity to run services. The lack of it will dwindle the resources that are working to keep the services up and running, and thus it impacts the service. As another example, on the capabilities front, knowledge about the services is like pure gold. If knowledge is well maintained, then the resolution of incidents or making changes is a hassle-free exercise. If the knowledge is all over the place or if the knowledge is not retained, then the real challenges creep in, with longer resolution times and malicious changes. Therefore, it is critical to note that all the service assets have a role to play in the delivery of services.

If there are any changes made to the service assets, then they need to go through the change management process. However, the change management process may not be common for all service assets. For example, making changes to a process may involve a quick conference between the process owner, process manager, business counterparts, and service managers. They sit down, discuss the proposed change and ratify it if satisfied. But, the underlying premise is that for each of the service assets, there must be a governance body around it (call it change management or anything else) to ensure that the changes made do not negatively impact the services.

Changes come in several layers such as strategic, tactical, and operational. Strategic and tactical changes are done at a level that's outside the scope of the project, be it an ITIL project or a DevOps project. Only operational changes come go through the change management process that I am going to discuss in the rest of the chapter.

Change in Scope

In Figure 9-1, I indicated infrastructure and applications in a different shade as the purview of change management comes directly under ITIL. Simply put, in the IT industry, change management refers to making changes to the IT infrastructure and the applications that contribute to the services. In the rest of the chapter and the book, when I refer to change management, I am talking about changes to be done either on infrastructure or on applications.

Note The official definition of *change* is as follows: “Change is the addition, removal, or modification of anything that could have an effect on IT services.”

Even after reading the official change definition, the scope may not spell out everything that you wanted to know if black and white. An IT service can spread far and wide, including the suppliers that support the service, the IT professionals who manage it, and the documentation for it. Does changing any of these peripheral components call in a change? Yes, but it depends on the agreement between the service provider and customer organization. Managing more items requires more time and resources, which adds up to expenses. If the customer wants to have absolute control over the IT services, then yes, every single element that makes up a service must come into the purview of change management. In the real world, this is often not the case, owing to the financials. Many of the indirect components are ignored in the interests of reducing expenses, and some companies find innovative ways of controlling the peripheral objects using standard changes and service requests.

There is much more to change management than adding, removing, and modifying IT services. Take the example of running an ad hoc report. You are not adding, removing, or modifying anything, just reading data from the database. Yet, you possess the power in your hands to break systems with the wrong set of queries that goes searching in every table, that utilizes the infrastructure’s resources, and that could potentially cause performance issues to the IT service. In this case, if you bring this through to change management, they can possibly identify the resource-consuming queries and shelve them or schedule them to be run during off-peak hours.

In the ITIL fiefdom, a change is painted as:

- Architectural changes to infrastructure and applications
- Additions, deletions, and modifications done to infrastructure CIs
- Additions, deletions, and modifications done to application CIs
- Database schema changes, migrations, data transformations to data used by services
- Configuration modifications performed on infrastructure and application CIs

The list is not comprehensive, but the underlying idea is that anything that matters in the running of a service is going to change, so it should be brought under the lens of the change management process.

Why Is Change Management Critical?

In Chapter 6, I mentioned that the service asset and configuration management process is a critical process and is the foundation for the rest of the processes to follow. While configuration management provides the foundation, the change management uses the foundational data to make decisions that could potentially break the existing services or could do wonders for the service. The change management process is a governance process that has complete control over what changes can be performed to the services and to the products that make up the service. If you get your change management process, the service management implementation is bound to get mature faster and make way for service improvements quite early in the cycle.

Change management is the sole authority that keeps a close watch on all the changes that are proposed, and after conducting due diligence and ensuring that all concerned parties are happy with the change, it gives its go-ahead for the commencement of the change. This ensures that all the stakeholders are made visible of the change and are given the opportunity to ask questions and oppose it. The process is responsible for bringing in a framework to ensure that changes go through a common pipeline that is governed by a change governing body called the *change advisory board* (CAB).

The process is that a governing body gives approval for technical teams to carry out the changes. They are a management body; therefore, they do not control the sluice gates if the technical team wants to make changes in a discrete manner. Such changes that are done outside the purview of change management are called *unauthorized changes*. Such changes hurt organizations, and service management generally loses control of their services because of such irresponsible actions by the technical teams. To state an example, an unauthorized change does not lead to changes to the content management database (CMDB). So, if an application is installed without going through the change management process, then the application CI does not get registered in the CMDB. In the future, when an incident is raised against the data coming out of the installed application, the application support teams refer to CMDB and become perplexed because there is nothing to troubleshoot. They need to later dig into the server to check what exists, and eventually, the overall diagnosis and troubleshooting take a lot more effort, leading

to extended downtimes and perhaps penalties. All such unwanted and unnecessary outcomes are a result of unauthorized changes. Therefore, it is critically important for the change management process to spread awareness on making the changes the right way and keeping a tight leash overall services in scope. One of the ways that the process keeps a watch over unauthorized changes is through the service asset and configuration management process audits, which are responsible for identifying such discrete changes. In the DevOps change management process, I will introduce a technical way to keep a close watch on the aspects of changes to the services and products.

A major mistake that most ITIL implementers do is to prioritize the design and implementation of the incident management process over the change management process. It's true that a process is needed for handling downtimes and degradations, but it is a whole lot more important to keep sentries at the door of services to allow only the authorized ones through. This blip of prioritizing the wrong process has put many ITIL projects on a path of constant firefighting even after months and years after the completion of the implementation. My advice is simple. Put a change management process in place to keep track of all the processes. This process may not be a full-fledged one but rather something like its skeletal cousin that can help prevent unauthorized changes. Once a full-blown incident management process is designed and implemented, get to work in bolstering the change management process.

Objectives and Scope of ITIL Change Management

The change management process is one of the governances put in place in the ITIL service management framework. It is a process that controls the changes that go into the IT environment. It is a process that acts as a gatekeeper, vetting, analyzing, and letting through only the qualified changes.

According to ITIL, the official definition of *change management* is that it controls the lifecycle of all changes, enabling beneficial changes to be made with minimum disruption to the services. This is change management in a nutshell: ensure only beneficial changes go through, remove the wheat from the chaff, and ensure that if anything were to go wrong, the risks are well understood, mitigated, and prepared for with minimal disruption.

To expand on this, the way this process works is somewhat like the changes that get funneled through the change management process. Only the changes that pass all the criteria set forth, that are of good quality, and that are deemed beneficial to customers

or service providers, in general, are approved for implementation. You should also know that the buck stops with change management for providing approvals. Implementing changes is managed through the release and deployment management process, which works in tight integration with the change management process. To reiterate, change management is accountable for providing approvals, and the release and deployment management process is accountable for implementation and post-implementation activities.

Digging deeper into change management, the output, or the objectives, of change management are as follows:

- Respond to the customer's ever-changing needs (technology upgrades and new business requirements) by ensuring that value is created
- Align IT services with business services when changes are planned and implemented
- Ensure all changes are recorded, analyzed, and evaluated by the process
- Ensure only authorized changes are allowed to be prioritized, planned, tested, and implemented in a controlled manner
- Ensure changes to configuration items are recorded in the configuration management system
- Ensure business risks are well understood and mitigated for all changes

Types of Changes

One size does not fit all the changes that happen in services. They come in all shapes and sizes. Therefore, you cannot use the same yardstick for all changes. You need a different set of protocols, policies, and processes to handle various types of changes. Say, for example, you trip over a water pipe and hurt your shoulder. You go to the hospital, and a doctor tends to you and does what is necessary with minimum fuss. Instead, if you were in a car wreck that required stitching you up after and putting some dislodged organs back in their place, this process will require an operation, surgeons, an anesthesiologist, and nurses among others to be present to ensure you survive and the operation is a

success. Between the two instances, the processes carried out is expectedly different, as one instance requires a host of professionals, utmost care, and some amount of planning, while the other can be done as and when needed with a basic medical skillset. For the trip-over patient, you don't need to assemble surgeons and others. Likewise, in change management, some changes need to be dealt with proper attention, planning, and care, while others can be carried out with minimum scrutiny.

In ITIL, there are three major types.

- Normal changes
- Emergency changes
- Standard changes

For your organization, you can define as many types of changes as you need. ITIL is not prescriptive, so the types of changes are served at best as a guideline. I once worked for an organization that had a fourth type called an *urgent change* that was placed between a normal change and an emergency change.

Type 1: Normal Changes

Let's say that a patient is ailing from a heart problem, and he needs to have open-heart surgery. The doctors and surgeons involved carefully and meticulously make all the plans, reserve the facilities, and then carry out the procedure. These are planned surgeries, and in the change management world, such changes that are planned in advance are called *normal changes*.

Most changes in any organization are normal changes as no organization wants to make changes without proper plans in place. Such changes are often lengthy because of the planning sessions, stakeholder visibility, and approvals, and to ensure that all the dependencies are managed.

Normal changes are generally associated with all the bells and whistles of the change management process and are often well analyzed, tested, mitigated, and verified. The maturity of an organization's change management process is often measured through the normal change process and the metrics and KPIs associated with it. Examples of normal changes include an application refresh to a newer version, a server migration from in-house to a cloud service provider, and the decommissioning of mainframe applications and servers.

Type 2: Emergency Changes

During his REM sleep, a man clutches his chest in pain and starts to sweat. An ambulance is called, and he is transported swiftly to a nearby hospital. The doctors diagnose a series of heart attacks that were caused because of a blockage in his heart. They don't have time to plan the surgery but rather do it right away if the patient is to survive. So, with minimum planning, they carry out the surgery. Such changes that are done during firefighting exercises are called *emergency changes* in the change management process.

Emergency changes are necessary to urgently fix an ongoing issue or a crisis. These changes are mostly carried out as a resolution to a major incident. The nature of such changes requires swift action, whether it is getting the necessary approvals or the testing that is involved. Generally, emergency changes are not thoroughly pretested, as the time availability is minimal. In some cases, they may go through without any testing, although this is not recommended, even for an emergency change.

The success of emergency changes reflects the agility of an organization and the change management process to act on disruptions in a time-constrained environment and to come out unscathed in the eyes of the customer and your competition. Emergency change management supports the incident management process in the resolution of incidents, especially major ones. Emergency changes are generally frowned upon and are not preferred. The number of such changes in an organization reflects poorly on the organization's stability of the services it offers.

Examples of emergency changes are the replacement of hardware infrastructure and restoring customer data from backup volumes.

Type 3: Standard Changes

A patient with failed kidneys gets dialysis done multiple times a week. The process for carrying out a dialysis procedure is well known and rarely can it go south. Most patients set up dialysis treatments at home and do them fairly regularly. The risks involved are low, and if something goes wrong, the impact too is on the lower end of the spectrum as there are multiple workarounds available. Such changes for IT services that pose no danger to services and are low key are referred to as *standard changes*.

Standard changes are normal changes that are of low risk and low impact in nature. I specifically use the word *can* as the categorization of changes as standard is at the discretion of the service provider and customer organizations.

Any organization will have a good chunk of low-risk and low-impact changes. In my estimate, it should run up to 50 percent of the overall changes or thereabouts. The service provider's responsibility to deliver Agile change management depends on their ability to identify standard changes from the normal change list and obtain the necessary approvals to standardize them.

Standard changes have distinct advantages and create value for customers. They follow a process that is less stringent and is free from multiple approvals and lead times that are often associated with normal changes. This provides the service provider with the arsenal needed to implement changes on the fly, which increases productivity and also helps deliver better value to the customer.

Examples of standard changes include minor patch upgrades, database re-indexing, and blacklisting IPs on firewalls.

ITIL Change Management Process

Of the three types of changes (normal, emergency, and standard), the normal change process is elaborate, it is lengthy, and it contains the most elements of the change management process. Figure 9-2 indicates a typical workflow for the normal change management process.

At first glance, it looks complicated, but as I break it down, you will understand that it is logical, and perhaps this will give you insight into how changes are recorded, approved, and implemented. The workflow boxes indicate process activities, and the text on the outside indicate the people/team responsible for carrying out the activity.

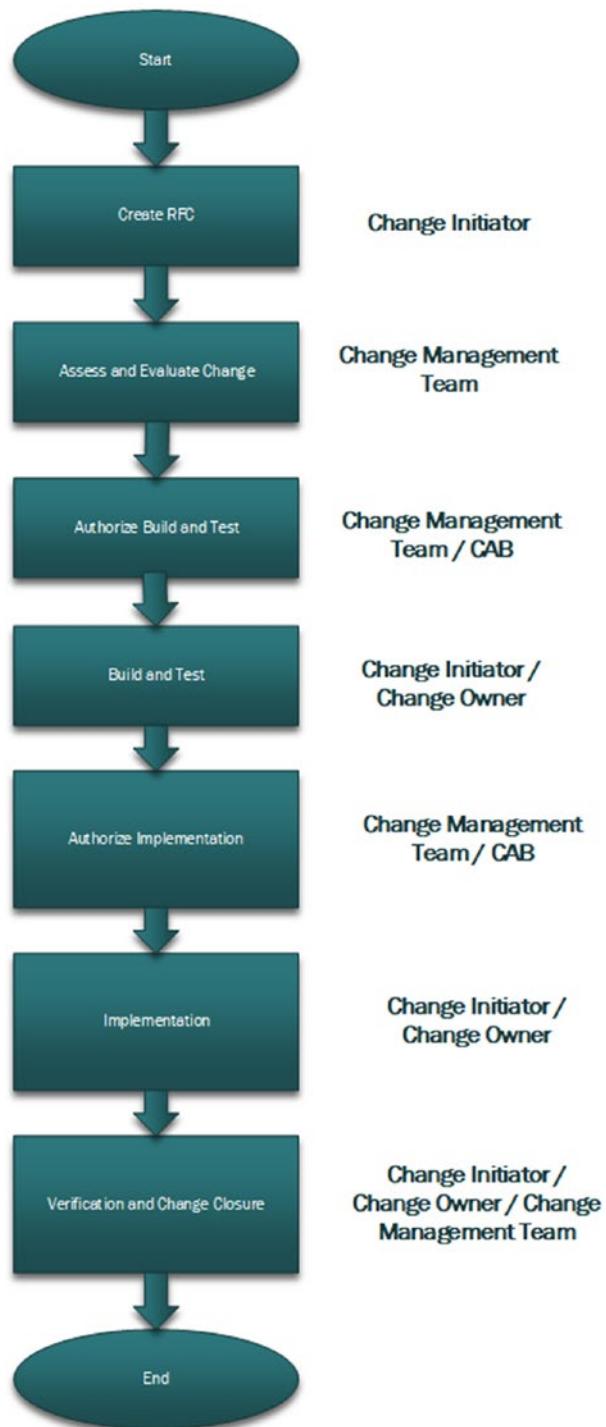


Figure 9-2. ITIL change management process for normal changes

Step 1: Create a Request for Change

A request for change (RFC) is a proposal initiated to perform a change. At this stage, the RFC is just a document with the change proposal. It is generally raised by the customer team or the technical team. There are no approvals or authorizations to perform the change.

The RFC document consists of all the necessary information pertaining to a change. RFCs will vary for every organization. The information needed, the format, the depth, and the necessary technical jargon are dictated by the change management policy.

Generally speaking, an RFC consists of the following fields:

- Change number
- Trigger for the change
- Change description
- Configuration items that are changing
- Change justification
- What happens if the change is not implemented
- Change start and end date and time
- Change category (major, significant, minor)
- Who is involved in the change
- Test plan
- Implementation plan
- Back-out plan
- Verification plan

In most ITIL change management implementations, the RFC is directly available on IT service management tools such as ServiceNow and BMC Remedy. The details required are collected using web forms. A few years back, before ITSM tools were used, RFCs were presented in the form of Microsoft Excel templates. IT stakeholders used the copies of the template to populate and send RFCs to change managers for processing and approval. This process was rather cumbersome as it was manual and was not governed using a system that applied the same yardstick for all change requesters. Change management has come a long way with the application of digital technology.

Step 2: Assess and Evaluate the Change

The RFC is analyzed and evaluated for risk, impact, and conflicts. The change management team is responsible for performing the assessment. They typically understand the change details, check whether the right stakeholders are indeed carrying out the change, and check for conflicts with connected systems and related changes going in during the same change window, among other conflict criteria. When the change is free of conflicts and is fully scoped and documented, it is scheduled to be presented in the CAB meeting.

In this activity, the role of the change manager is critical as the change manager alone would have visibility across the organization's changes and are in the best position to identify conflicts, if any arise. For fantasy fanatics, you could think of change managers as the first line of defense against potential malicious changes.

I worked as an enterprise change manager at one stage in my career. The role was daunting, and knowing that the entire billion-dollar organization depended on my foresight and analysis was a scary thought. It was a challenging role that I enjoyed during my heyday in an operations role.

Step 3: Authorize the Build and Test

The change manager calls for a CAB meeting of all stakeholders, from the technical and business lines in the organization.

In the CAB meeting, the change manager leads the meeting and presents the change. The CAB provides its approval for building and testing the change. The forum provides authorization for developing the change, and this is the most critical approval in the change management process.

Change Advisory Board

The CAB exists to support the change management team and to make decisions on approving or rejecting changes. To state it simply, it can be described as an extension of the change management role, and it exists to ensure that the proposed changes are nondisruptive, scheduled to minimize conflicts, prioritized based on risk and impact, and analyzed for every possible outcome to the hilt.

In an organization, you can have multiple CABs to support change management. A typical example would be a change being represented in an infrastructure CAB before it goes into the enterprise CAB and perhaps followed by a global CAB. The essence of having a CAB is important, not the way it gets implemented.

It is critical for the change owner to present the complete change to the CAB, with all the possible details. This will help the CAB decide on authorization to proceed with it. The CAB has the authority to ask for additional information to be gained, additional tests to be conducted and presented to them, and changes to be rescheduled. In some cases, the CAB can reduce the scope of the change to ensure minimal impact for business and technical reasons.

Composition of the Change Advisory Board

The composition of the CAB consists of stakeholders from the business as well as from service delivery. It can also include suppliers, legal experts, business relationship managers, and other stakeholders as identified by the chairperson.

CABs are dynamic. They could be different for every change that comes up for discussion. For a particular change, you may have Supplier A, a network manager, an Exchange manager, and IT security. For another change in the same CAB meeting, you may have Supplier B, an application delivery head, and the SAP manager as CAB members.

Some organizations might insist on a set of permanent members of the CAB who sit in on all proposed changes, during every single CAB, and additional approvers (dynamic) would come and go as necessary.

No matter who sits on the CAB as an approver, the change manager, who is responsible for all the change management activities, is the chairperson of the meeting and decides on CAB members, on the changes that get represented on the CAB, and on the final decision of the CAB.

These are potential CAB members:

- Change manager as chairperson
- Customers
- Suppliers
- IT security
- Service owners

- Business relationship managers
- Application delivery managers
- Operations managers
- Technical subject-matter experts
- Facilities managers
- Legal representation

Emergency Change Advisory Board

Emergency changes require urgent attention and quick decisions. A CAB will not work for assessing emergency changes. These changes may happen in the middle of the night and require people to spring into decision-making mode to approve or reject changes.

The need of the hour to help change management decide on approvals is the emergency change advisory board (ECAB). The need for emergency changes pops up through incidents. It is possible that carrying out an emergency change (unsuccessfully) could impact the service more than the incident itself. So, in all necessity, there is a need for a few extra pairs of eyes to look at the proposed emergency change and provide the approval in the most awkward hours of the night (or day).

In most cases, a change ticket will not be created when approvals are sought. Change documentation may be done retrospectively for emergency changes. So, it is imperative that emergency changes are approved based on what is heard and what was relayed.

An ECAB is comprised of key members who provide their decision on the proposed change. ECABs mostly happen over a phone line, and it's extremely unlikely that there would be the luxury of members sitting across from one another. In some instances, the ECAB members may provide their approval individually but not while they sit in a gathering. Individual responses are collected, and the change manager uses wisdom in providing a direction for emergency changes.

Note The changes that can come to an ECAB have specific, predefined criteria attached to them, such as a major incident requires executing a change for a fix that might have a significant impact to the organization or customers.

Not all emergency changes call for an ECAB. Most of these are approved directly by the change manager if the ECAB rights are delegated. The critical ones, where entire enterprises could possibly be negatively impacted, would call for an ECAB to make decisions. The emergency change management process should not be exploited to push nonemergency changes under the guise of emergency changes.

It is possible that ECABs could ask service delivery teams to convert an emergency change to a normal change if a workaround exists to keep the service running. An example could be a database containing customer information that has gone corrupt and the database team wants to restore data from backup tapes. They want to do it as an emergency change to ensure that the customer's data are present before the customer's business starts in the morning. An RFC for performing an emergency change is raised. An ECAB is convened, and approvals are sought. Change documentation may be done retrospectively. The database team restores the customer's data from the backup tapes, and the emergency change is a success. During the business hours, the change document gets created with all the bells and whistles, and it goes through the entire cycle of obtaining approvals for visibility and to keep other stakeholders who were not involved in the ECAB process informed.

There is a specific place for ECABs, and they have a specific job to do. This process must not be abused with trivial emergency changes knocking on the doors of ECAB. It dilutes the process and forces the ECAB members to lose focus on what really requires their attention.

Standard Change Advisory Board

The standard change advisory board (SCAB) is my own making; you will not hear or read about it elsewhere. A SCAB is a type of advisory panel that makes decisions on whether certain changes can be standardized. It is not a panel that works in the operations layers or service management but rather in the tactical.

The SCAB's prime objective is to ensure that the change presented to the board for standardization presents no major business impact if things were to go south; it also ensures that the possibility of things going wrong is minimal. Essentially, the SCAB is giving the technical team a free pass to carry out the standard change as per the agreed triggers with minimal external supervision. Therefore, it is important that the make-up of the SCAB consists of people who understand their areas very well and the associated business impacts.

Step 4: Build and Test

The build and test process, which includes software development, unit testing, system integration testing, and user acceptance testing, is not part of the change management process. I have included it under the change management process to provide continuity in the process activities. These activities belong to the release and deployment process, which is discussed in Chapter 10.

Step 5: Authorize the Implementation

The test results are presented to the change governance body, meaning change management and the CAB. Based on the results, the change management team provides authorization to implement the change in the production environment.

This is yet another important activity, as all the identified testing activities must be completed successfully before the change is allowed to be implemented into the production environment.

Step 6: Implement and Verify

The implementation and verification process is not part of the change management process per se. Like the build and test activity, this one comes under the release and deployment management process.

In this activity, the technical team will deploy the change in the production environment during the approved change window and perform post-implementation verification to ensure that the change is successful. If the change is not successful, it will be rolled back to the previous state if possible.

Step 7: Review and Close the Change

A post-implementation review (PIR) is conducted to ensure that the change has met its objectives. During this review, checks are performed to identify whether any unintended side effects were caused. There are lessons to be learned from changes. If there are any such candidates, it is fed into the knowledge management database (KMDB).

After the successful completion of the PIR, the change ticket is closed with an appropriate status, such as implemented successfully, change rolled back, change caused incident, or change implemented beyond the window.

The responsibility for carrying out this activity generally falls to the change management team, but some organizations have the change owners and change initiators close the change with the correct status.

How Are DevOps Changes Different from ITIL Changes?

When I worked as an enterprise change manager, I often got feedback from various delivery teams and business teams stating that the process was rigid, and they had to plan at least a few months earlier for major changes. When the execution of change approvals started, the number of stakeholder identification and approvals was cumbersome. Some even shared with me they decided to forego certain changes because of the lack of time and the enormity of the bureaucracy that was instilled in the process. I was managing changes for one of Australia's biggest retail outlets, and with a number of moving parts including presence of legacy applications and the lack of a proper CMDB, the change management process indeed asked for good amount of lead time for the analysis to be completed before the change went in for approvals, and it had multiple approval stages because we did not have a matrix (usually generated through CMDB) to identify the right stakeholders.

This was a pure service management project with no principles of DevOps inculcated in it. Hypothetically speaking, if I were asked to redesign the change management project as the retail organization decided to jump fully into the Agile and DevOps world, I still would not compromise on any of the change management principles. There is nothing wrong with it; there is a reason why certain principles exist like having a CAB look at the major changes and providing lead time before a change can be executed. What needs to strengthen, however, is the enablement of accurate information through a healthy CMDB and use of common sense and logic to cut down on unnecessary bureaucracies. Changes are in fact one of the main drivers for DevOps. Changes are good. The Agile methodology embraces changes and does not shy away from it. The change management in DevOps should be a steering process more than a governance process. It must help steer the entire DevOps boat through automation, simplification, and common sense.

The Perceived Problem with ITIL Change Management

At the core, the ITIL change management process is designed to manage complex changes. It calls for all the bells and whistles in terms of governing changes, from planning to ensuring minimal risks while making a change. The underlying unstated motto is maximum and effective governance with minimal risk appetite. However, the ITIL process also tries to be nonprescriptive in suggesting that the organization implementing the process can decide on its risk appetite and the level of governance scrutiny over the changes.

Organizations that tend to go by the book prefer to take a risk-averse approach and bring in all the changes under the same umbrella (following the same process for datacenter migrations and periodic security updates on a server). This is, in fact, a good approach for an organization that has stepped into the ITIL world. But companies mature, they must diversify and put changes of different magnitudes and impacts into different silos, with a different governance structure. This is frequently missed in most organizations today. Therefore, there is a sense of antipathy toward the change management process as a whole.

Change managers and people governing changes are considered people who are against innovation, blockers toward making improvements, and bureaucratic and traditional by mind-set.

The ITIL change management process is believed to be a sequential process, as you have a set process where certain activities have to be complete before you can embark on the next set of activities. The sequential nature is in stark contrast to what we are trying to achieve today, which is to develop and progress through iterative changes. Yes, the change management process is sequential. You cannot analyze a change if the change performed is not fully documented along with the risks it introduces. A decision on whether a change should go through cannot be made unless all the ducks are in a row.

There is a reason for change management's sequential nature. How can you build a roof if the foundation and the pillars are not erected? It's a fool's dream to make changes either with no governance or by doing activities in parallel. This is where applying DevOps methodologies brings the focus required to better the process toward progression and agility.

DevOps to the Rescue

When you are faced with a problem and if you kept staring at it from a unilateral perspective (such as change management being sequential and DevOps being iterative), the solution can be evasive. So what you need to do is to think outside the box or look at the problem from a different perspective. Such as, don't concentrate on the solution to a problem but rather take a step or two back, and understand what the objectives are that you want to achieve. Find a solution to meeting the objectives rather than the problem itself.

The approach DevOps has adopted is to make frequent changes with absolutely minimal governance. If something goes wrong (even terribly), the impact will still be manageable, and rolling back will be brisk as the rollback too will be quite straightforward (given that the change is a small one). When you do such multiple changes on the back of one another, you are managing the risk effectively, and the combination of the multiple small changes is probably equivalent to a decent-sized change (or even a major one). And you are able to deliver changes with (almost) no lead time and with minimal governance and manage risks that come as a result of its failure.

DevOps changes the nature of change management from being sequential to iterative. Within iterations, change management is still sequential, but because of the scale of the change, its sequentially does not become bothersome.

DevOps can be applied to projects that are built in an iterative model alone. For example, datacenter migrations or nonsoftware projects may find it difficult to follow the DevOps iterative approach to change management.

Project Change Management

Earlier in this chapter, I introduced the three types of changes in the ITIL world. When we look beyond ITIL and service management, there are multiple types of change management. I am not referring to the strategic, tactical, and operational changes. In project management, we refer to change management as *change control*, where changes pertaining to the triple constraints (namely, scope, cost, and schedule) come under the purview of the change management process. In fact, it is considered that quality is the fourth constraint, and this too comes under change management's purview.

Any changes to scope, cost, time, and quality go through the process of change control. It is a fairly bureaucratic and time-consuming process that considers changes to any of the aspects and measures the other constraints based on the change. This is a fairly common process that gets implemented and practiced religiously in projects that follow the waterfall model of project management.

In the Agile world, although some projects do maintain this process and practice, I do not see its value. As the foundation of Agile project management stands on being Agile and embracing changes, keeping a change management process to manage the constraints sounds hypocritical. Let's say the process exists in an Agile project, and considering that changes are fairly common, at the beginning of every sprint, you might expect a formal change control process to kick in and sort out the impacts to other constraints, but to what effect? It is going to change again anyway. You are going to do this all over before the next sprint begins. The efforts that go into managing the changes will probably be a good percentage of the overall project management efforts, which is a waste.

But then, how do we manage projects if we have no control over the changes that are flowing in? The planning of projects is done at a fairly high level, and the actual planning of development is done during Agile release trains (ARTs) if SAFe is the Agile framework or at a sprint level for a Scrum-based approach. At the ART or sprint level, we know exactly what we are trying to achieve, and all the planning is done during the sprint planning session. The quadruple constraints are managed in the following manner:

Scope: The scope of a sprint is managed by the product owner, who is preferably from the client organization, and he identifies the set of user stories that will be acted upon during a particular sprint. This is a classic example of the client having complete control of the scope and cherry-picking the functionalities that need to be developed on priority.

In ART, multiple product owners and relevant stakeholders are involved during the program increment (PI) planning session held over two days. During the planning session, the set of functionalities that are to be developed over the next ten weeks is charted with ample wiggle room for changes if any during this period.

Cost: The costing model for an Agile project has been a discussion that has sparked multiple conferences around the world as well as garnered energy and enthusiasm in digital and physical forums. The question is, if you are not clear about the scope, then how do you set a budget and track the costs of a project? Since the scope is a moving target, the costs too will start to shift right, making it impossible to manage. While scope being moved repeatedly is okay with most organizations, when it comes to finances, the freedom is measured differently. Organizations make yearly

budgets and don't like the accounts to swing too far toward the red, and it is believed that Agile projects can completely blindside the financial aspects.

Models for budgeting and accounting projects can be created in a number of ways. Any project can be logically dissected into logical phases to track and manage discrete bodies of activities separately. We generally don't start with sprints from day 1, as there are usually project inception and road map planning phases that precede development and testing. Resource allocations follow the pattern of project phases with business analysts, architects, and environment engineers preceding the development team in projects. Figure 9-3 shows the IBM Rational Unified Process indicating the various phases of a project and the allocation of resources against each phase.

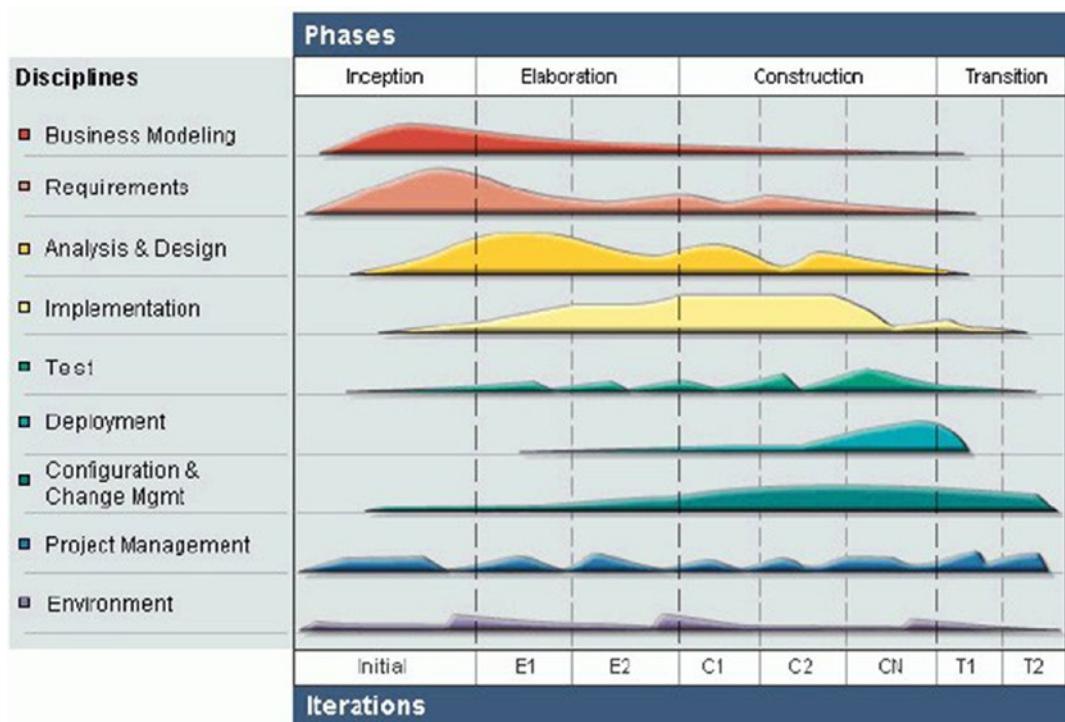


Figure 9-3. Resource allocation in a project (image credit: ibm.com)

For resources who sit outside the sprint, based on the resource allocations, budgeting and costing can be done in a fairly direct manner. It gets easier for teams in sprints. A sprint is made up of cross-functional resources, and the sprint make-up does not normally alter through the course of a project. Therefore, the cost of resources for each sprint is a finite number, and multiplying it by the number of sprints gives a fairly accurate cost of the developing and testing efforts.

Schedule: A waterfall project is a sequence of activities where the schedule is the direct result of all the sequential activities. The schedule is more likely to be extended than the scope. If more money is pumped in, the schedule can be squeezed by bringing in additional resources. In a lot of ways, in Agile project management, similarities exist in how the schedule can be managed through scope and costing. In a similar vein, scope and costs too can be managed by managing the other two constraints.

In an Agile project, the duration of the project can be accurately estimated not at the inception phase but rather after running through a couple of sprints. The sprint team's velocity can be determined after a couple of sprints. Determining the velocity and factoring it against the product backlog may give you a clue on the number of sprints that are needed to clear it. However, in Agile projects, the product backlog keeps churning all the time. At the end of a sprint, the demo might throw up several changes to the developed functionality, and some functionalities in the product backlog could be replaced by other complex functionalities that become the need of the hour. So, I would be surprised if anybody can put a finger on a number to indicate the duration of an Agile project.

What we can predict, though, are the number of the story points that can be delivered over a period of time. Story points are a unit of velocity, and through velocity determination, we can come up with the number of story points that can be accomplished in the next few weeks and months. If a project is realizing a product launch (let's say), then the product owner (PO) is at the helm of

affairs to decide on which user stories from the product backlog make it to the sprint backlog. The PO has been intimately aware of the velocity and can make decision calls to move priority items into a sprint.

Quality: Quality, the fourth constraint, is a critical aspect of any project. It has a direct correlation to the cost of the project as the costs tend to increase as we try to increase the quality.

Likewise, the schedule of a project tends to get extended if a high benchmark is set in terms of the quality to be achieved.

With DevOps coming into play, the game has changed. It is no longer just the project management aspects that alter the quality landscape. DevOps pipelines can ensure that the right quality is maintained in the software based on the set expectations. The best part is that the pipelines have to be set up once (with minor tweaks in between, of course), and the software can be run against the test scripts in an automated manner as many times as needed without precious human resources. This setup has given a strong backbone to projects to not worry too much about quality pushing the schedule ahead or exceeding the budgets toward deep red. As long as the quality is well thought through and test scenarios and scripts target the heart of user stories, the quality of the software can stand on its own, even amidst the ruins.

Risk Mitigation Strategies

The change management process exists to minimize the risks of untoward changes going into the system and to make the stakeholders aware of the possible risks by going through with the change so that the stakeholders can use their discretionary powers to either allow or deny the change. That said, the change management process prefers to go with approaches that are less risky.

DevOps is just not a methodology that automates everything and quickens the software delivery lifecycle. There is more to it than the general DevOps-aware public knows. One of the main tenets of DevOps is to uphold the quality of software, and the quality has a direct correlation with the risk each change imposes. The lower the quality, the higher the number of risks, and vice versa. Not only in the software industry but in all industries, the lack of quality is one of the top-rated risks, but by using DevOps methodologies and processes, organizations can ensure that high quality becomes a hygiene factor rather than something that they strive to achieve.

Auto-Deployment and Auto-Checks

Automation kills two birds with a single bullet. The first and fairly obvious one is that automation is far more effective than humans, and productivity is bound to increase as the lead time from one activity to another is typically nonexistent. The second point I am making in this section is that automation eliminates human error.

Humans are imperfect; we have moods and emotions and are open to distractions. The activities we perform in the software delivery lifecycle are open to errors, even if it is a simple task of copying files from one location to another. Simply put, humans cannot be trusted to carry out work with zero errors because of carelessness, confusion, or ignorance. These shortcomings can be mitigated by asking systems to do it for us, and if the design of automation is perfect (which is a one-time activity), then the machines can carry out the job entrusted to them time and again error-free.

Automation does have its drawbacks. It does not have its own intelligence (duh!), and the activities that it can perform are merely repetitive ones that we have identified, designed, and implemented into it. These are some of the repetitive activities in a software delivery lifecycle:

- Code review
- Code build
- Artifact storage
- Artifact retrieval
- Testing (all types including acceptance)
- Defect logging
- Deployment
- Reporting

As I discussed earlier, the quality of the software is greatly enhanced by applying automation with the right rigor, which correlates with various testing activities. However, what goes unnoticed is the risk mitigation through the other automation that we achieve, primarily in the areas of code reviews and deployments.

Software projects can reduce risks (and make change management happy) by employing automation to the fullest extent. In my experience, apart from the defects, manual deployments on the production environments gave rise to a number of failed

changes, which can be easily mitigated through automating deployment by using tools such Ansible and Puppet. The automation design around deployments is straightforward and simple compared to the automation around testing, so projects must ensure that all their deployments happen automatically, giving no leeway for human errors to creep in.

Static and dynamic code reviews give an insight into the health of the code and the binary. By automating code reviews using tools such as SonarQube and Crucible, any risks emanating based on unhealthy coding practices and faulty binaries can be mitigated through a simple integration of the toolsets into the DevOps pipelines.

DevOps Change Management Process

I have heard from quite a few hard-core DevOps proponents that there is no room for the change management process in a DevOps-driven project. They say that things move way too fast for any approvals and bureaucracies to creep in. When questioned on the risks it imposes and the lack of transparency, I was told that the magnitude of changes was so small that most times even if things were to go south, they could fix it by any means necessary as long as they didn't have to report to any governance bodies. This answer did not bode well with me. Although I am a big advocate of moving to DevOps, implementing without a governance body, especially without a change management process in place, is a disaster in making. This particular incident was one of the reasons for the existence of this book.

Earlier in this chapter, I introduced the ITIL change management process, which had at least a couple of “bureaucratic” blocks (if I may say so) that required authorization before moving forward. The rest of the activities mentioned within it were logical for any project in terms of coding, testing, and implementation. The ITIL change management process has been proven time and again to be effective against malicious changes, so my effort here is to incorporate the process in a DevOps project without sacrificing the process objectives and, at the same time, not introducing hurdles in the DevOps way of working.

In Chapter 1, I introduced the two main processes of DevOps that are leveraged to provide end-to-end delivery of software from the coders' desk to the production boxes: continuous delivery and continuous deployment. The change management process is adapted differently for each of the processes.

Change Management Adaption for Continuous Delivery

Typically in projects governed by ITIL change management, technical teams develop and test all their wares before approaching the change management process for approval to implement. As the ITIL change management process recommends a two-step approval and authorization, first from the CAB and later from the change management team, the process is looked at as hindering the good work done by the technical teams and as obstructing the complete value by implementing it at the earliest. This is a criticism from the ignorant, but when we try to adapt the same change management process in a DevOps project, it is important to emphasize the activities that need to be done even before the first line of code is written.

The change management process adaption for continuous delivery is indicated in Figure 9-4.

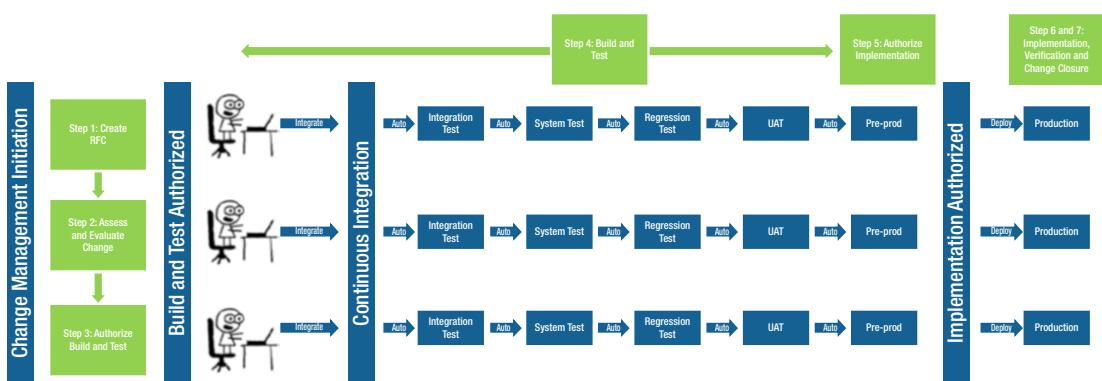


Figure 9-4. Change management adaption for continuous delivery

Continuous delivery is a development and deployment process where all the development and testing activities are automated, except for the deployment to production. The deployment to production requires a manual trigger for the binary to hit the production boxes.

Steps 1, 2, and 3: Change Initiation

The process of continuous integration where coders check in their code frequently, followed by automatic build, review, and the test, should not and does not change. However, as I indicated earlier, change management wants to know what you are doing before you start doing it.

Before the coding begins and well after the requirements are documented and agreed on and designs and blueprints are drawn up on paper, the change management process must be initiated. Initiation of the process begins with the process to raise a request for change (RFC), indicating that you intend to make changes to the existing production environment, either through the introduction, modification, or removal. The RFC is analyzed by the change management experts and the relevant stakeholders, and the members of the change advisory board are identified. The CAB is convened, and the owner of the change is summoned to the forum to present the change and answer all the questions the CAB members might have. The CAB will provide its approval when the change owner is able to convince them of the value that the change brings in, along with the risks identified and its mitigation actions.

Step 4: Build and Test

After the CAB approval/authorization, the build and test activities can start; this is continuous integration. The continuous delivery process ensures that the coding and testing of the package are done at a level that the binary can be deployed at any time, meaning it is ready from a quality standpoint. The activities surrounding build and test in the change management process is correlated to step 4.

Step 5: Deployment Authorization

When the binaries are ready to be deployed, meaning the coding and testing activities for the scoped set of requirements are completed, the change owner goes back to the change management team indicating that the binary is ready to be moved into production. The change management team carries out some basic checks to ensure that the scope of the binary is as agreed on in the RFC and that the timelines for deployment are acceptable to all stakeholders. When all checks come out okay, they give their authorization for implementation, as indicated in step 5.

Steps 6 and 7: Deployment and Verification

The binary is deployed to production during the agreed upon change window. Smoke tests and other verifications (post-implementation review) are done as a part of ensuring that the change is a success. Once it is deemed successful, the change ticket is marked up as a success and the ticket is closed. These actions are indicated in steps 6 and 7 in the change management process.

Continuous Delivery for Maximum Change Governance

The continuous delivery process is perhaps what the change governance prefers as the process ensures that the production systems are untouched unless the proposed changes are tabled back to the change management team. This ensures that the deployments to production are governed with hawk eyes, and this provides a sense of control for the change management governance teams.

Continuous delivery is a good process for most organizations, as they are able to draw a good balance between control and automation. The productivity is unaffected as the entire chain of development and testing activities is running smoothly on the back of automation; during deployment, the governance comes into play to decide on further courses of action.

Change Management Adaption for Continuous Deployment

Continuous deployment is the big brother of continuous delivery. It is a lot older and more mature and prefers to take an automated approach. There is no manual trigger for binaries to move into deployment. As soon as a piece of code is developed and tested satisfactorily (as determined by machines), it gets deployed onto the production boxes automatically. There is a pause to see whether everything is okay because the changes being done are minute and chances are such that nobody will even notice them. It is like an army of ants moving sugar crystals one at a time to its home, and over a period of time, a sack of sugar which was brimming is now only half full. The movement happened over a long period of time, continuously, and since the volume of the movement was so low, nobody even noticed it. After a couple of months, though, the change is visible to the naked eye. Likewise, continuous deployment changes happen in minute batches, and the collective set (say around 10,000 such changes) might represent a change that a project employing continuous delivery opts as a change.

Although most organizations have trepidation toward continuous deployment, it is not risky in the sense that it might break systems or it is an accident waiting to happen (because of a lack of governance). It is similar to a jigsaw puzzle being constructed, one piece at a time, and over a period of time, the entire picture comes together. If one of the pieces don't fit into a particular groove, no problem; it is pulled out, and adjustments are made.

In Chapter 1, I explained continuous deployment in detail. Figure 9-5 reminds you of the ITIL change management process. Did you just hear me right? Yes, you can still apply and adapt the ITIL change management process for the continuous deployment process. But how can you govern when everything is automated? Everything is automated after the development starts, but there's a whole lot of open space before and after.

At the outset, when you compare the change management adaption for continuous delivery and the change management adaption for continuous deployment, they look similar. But there is some shuffling of the change management process to fit the automated nature of the continuous deployment process.

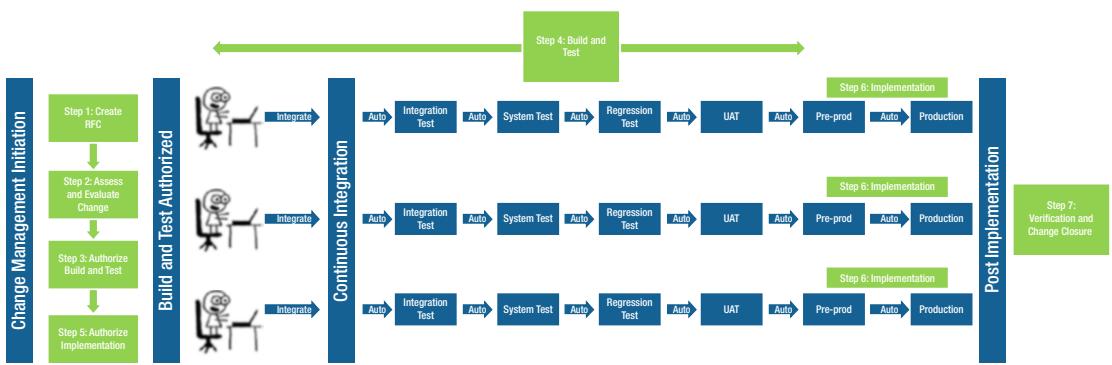


Figure 9-5. Change adaption for continuous deployment

Steps 1, 2, 3, and 5: Change Initiation and Authorization to Deploy

The change initiation steps of creating an RFC and running through the CAB is similar to the continuous delivery process. The rules remain the same; there is no coding and testing to be done unless the CAB gives their go-ahead. But there is a difference in what you seek from the CAB.

When the change owner presents the change in the CAB, the change owner proposes that the CAB provides not only the approval for coding and testing but also the authorization to implement the binaries as and when they are deemed successful, generally multiple times a day. The change owner has to convince the CAB that the quality will not be compromised by sharing the DevOps pipelines strategies, plans, and various checks and balances put in place. This change presentation will be a lot more rigorous than the one faced for the continuous delivery process.

The CAB will provide its approval only if it thinks the pipeline that has been constructed meets all the quality standards put forth and does not come in the way of accepting more risk than necessary. Yes, once the approval and authorization are obtained, the change owner gets a VIP card to the production systems that let him deploy as many changes as needed during the duration of the change, as specified in the RFC. The CAB holds the authority to pull back the VIP card if the failure rate of changes exceeds a certain threshold. Once the card is pulled back, handing it back requires additional plans, tests, and quality checks to be put in place.

Step 4: Build and Test

For developers and testers, the coding and testing activities remain the same, whether they work in the continuous delivery process or the continuous deployment process. The principles remain the same: to innovate, experiment, and learn from mistakes.

Step 6: Deployment to Production

In Figure 9-5 , you will notice multiple steps as the deployments happen multiple times. The DevOps pipeline is built in such a manner that if the checked-in code is successfully built and satisfactorily tested across all the various designed tests, the binary gets deployed into production automatically.

If any of the tests were to fail, then the binary will not get promoted to production, but rather a defect is raised for the developer to fix and rebuild the code. This is a powerful way of ensuring that the quality is not compromised even in the wake of full automation; it's a hands-off approach. These quality parameters have to be built into continuous integration orchestration systems such as Jenkins and Bamboo.

Step 7: Change Verification and Closure

When binaries are deployed into production, the verification activities (smoke tests) are automated as well. The automation ensures that the deployed binary is working the way it should, and any anomalies should trigger an automatic rollback of the deployed binary.

The change closure happens when all the objectives of the change are met. The change management team will dive deep to identify what changes have been done and whether there is scope creep. In cases where scope creep is identified, the change owner is reprimanded, and the opportunity to leverage continuous deployment (VIP card) is not entertained in the near future.

Maximum Agility with Standard Changes

There's another way of achieving flexibility and agility in the world of governance in service management: standard changes. Standard changes are changes that are pre-approved to be executed during certain scenarios. The changes that are categorized as standard usually don't bring down entire systems or cause fatal damages; they are low risk and low impact. Most important, these changes are documented with specific procedures.

It is believed that the maturity of the service management process of organizations offering services can be determined based on the number of standard changes in the system. That's true, as standard changes present a system to segregate difficult changes from the usual ones. The commonly performed changes are like a well-oiled machine. It operates smoothly and can be relied upon in most circumstances. Around 60 percent to 70 percent of the changes in any organization are common, repeatable, and straightforward. If all these changes are standardized, imagine the number of approvals that don't have to be sought and the number of meetings, telephone calls, and waiting around that can be skipped.

The advantage with standard changes is such that in most situations, it can be done on the back of any defined trigger. When a team wants to perform a standard change, they don't have to go to the change management team or to the CAB to present their change. They simply log a standard change in the system (yes, records are an absolute necessity), and then they can carry it out. Once it is successfully implemented (which is expected), the change is closed. *Voila!* There's no need to do a post-implementation review.

Examples of standard changes can be anything and everything under the IT sun that is repetitive in nature and does not pose major risks. That sounds like every single deployment we do under DevOps, doesn't it? Do you now see the connection between standard changes and DevOps? Typical examples include installing security patches on operating systems, running batch jobs, and performing nonintrusive backups.

Championing Standard Changes

I started my career as a service management consultant, and over the years, I earned the reputation of creating value for my clients through my designs and improvements. Implementing standard changes was one of my secret weapons. These are the first things I look at during an assessment:

- Does a robust change management process exist?
- Are there provisions for standard changes?
- How many changes are implemented as standard changes?
- Are standard changes monitored and audited regularly?

Standard changes are the low-hanging value creation fruit for clients. Most service management experts and consultants have yet to come to grips with it, and that hurts their clients' chances of making a difference through service management.

When I worked as an enterprise change manager for a retail organization in Sydney, Australia, I noticed that the organization did not have an active standard change process. A team of change managers reviewed and processed anywhere between 150 to 200 changes each day. They knew the changes so well that by reading the change summary, they would just scroll down to where the approval task became visible and hit Approve. What struck me first was why this human effort was even required as it amounted to a process to be followed rather than any visible value through the additional pair of eyes.

I got down to work by pulling data for the past couple of years and identifying such changes that could potentially be categorized as standard changes. I did some arithmetic and some guesswork to crunch some numbers, and according to my analysis, the organization could save efforts anywhere between 25 hours and 40 hours every single day. I considered modest numbers for my calculations, and this was the minimum savings that the organization could do. Considering 200 hours of weekly savings, and the average wages for a week typically in Australia is about \$2,500 (\$12,500 for the 200 hours saved). This saving translates to a monthly savings of around \$60,000 (\$12,500 X 4 weeks). Out of nowhere, the company could save over half a million every year, and they jumped onto it, not without a number of warnings from the company old-timers.

I managed to convert about 60 percent of the overall changes into standard changes within four months, and the benefits of showcased the power of Agile and DevOps. A number of businesspeople in this organization were relieved not to have to worry about getting approvals for all their changes. The best part was that they didn't have to bundle their changes into releases, so the changes went quickly to production and provided maximum benefits to the business.

Process for Identifying and Managing Standard Changes

There is no guideline or a typical process in the ITIL publication to identify and manage standard changes. I believe the reason could be that standard changes are viewed as a mature service management element and the ITIL publication has provided guidance on processes and procedures around generic service management processes only.

I devised the process that I present in this section, and it has been implemented across organizations with excellent results.

In this process, steps 1 through 4 are employed sequentially for identifying standard changes, and steps 5 and 6 are independent activities for managing standard changes and are not carried out in any sequence. In fact, steps 5 and 6 are processes on their own (I will provide a glimpse of how the process activities can be designed under the process steps).

Figure 9-6 illustrates a process for identifying and managing standard changes.

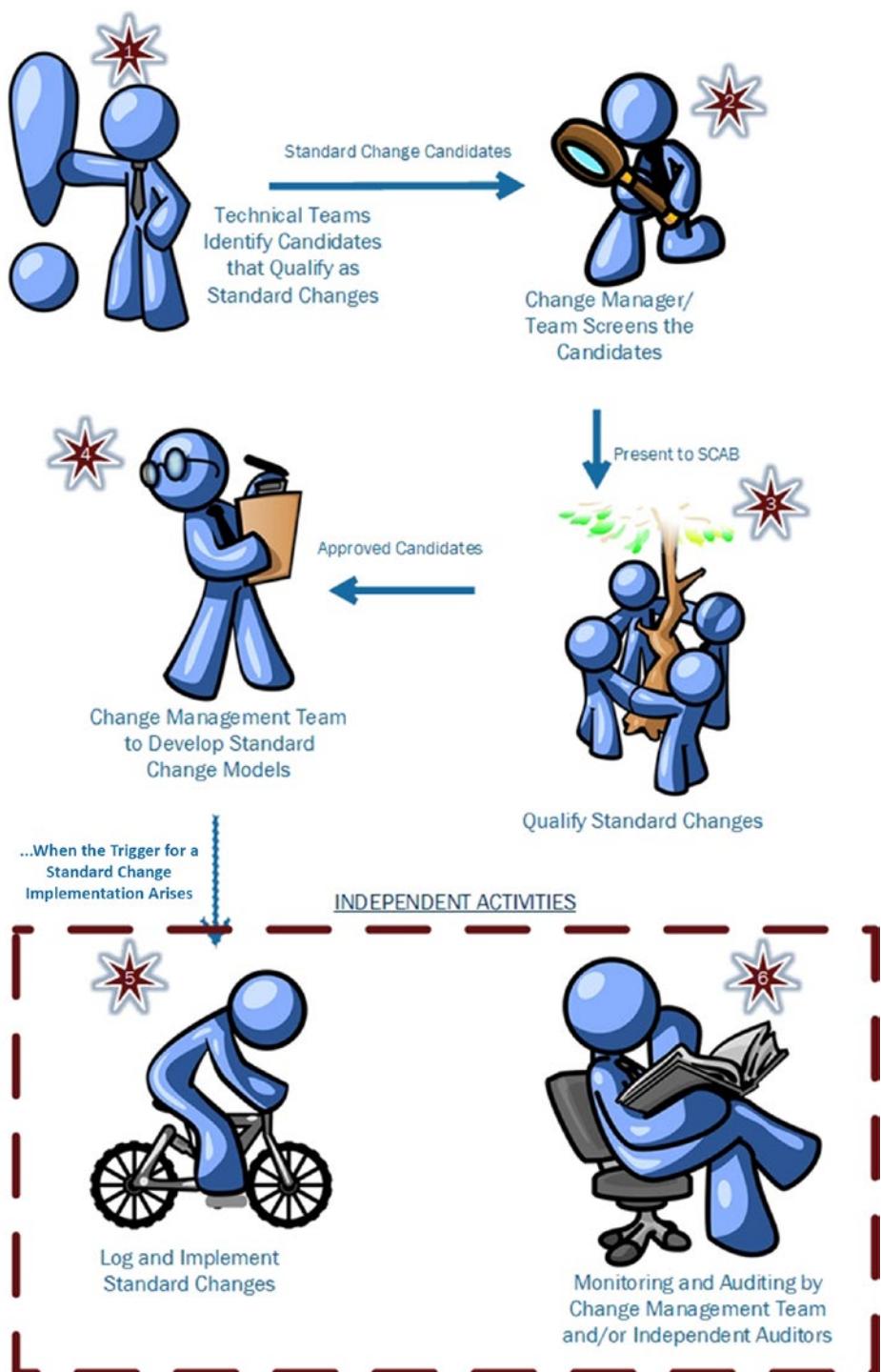


Figure 9-6. A process for identifying and managing standard changes

Step 1: Identify Standard Changes

The best team or people to identify standard changes are technical teams themselves. They best know their line of work and the changes they have been carrying out for days, months, and years. So, they will be in a great position to qualify those changes that are of low impact, that are low risk, and that are carried out multiple times in a given period. For example, if there was a super-simple change that gets carried out, maybe once every two years, then there is no point in standardizing it as the effort required to standardize it is far greater than implementing the change through the normal change process.

The process for nominating candidates can be done in a number of ways. The simplest one is to manage a Microsoft Excel template that asks for qualifying details to be filled out such as the following:

- Summary of the change
- Justification on low impact
- Justification on low risk
- Number of times the change might be implemented in a given period, say a quarter
- Trigger for the change (under what circumstances will it be implemented)
- Document the steps were taken to implement the change

In some of my implementations, I did not use Microsoft Excel but instead used the change management module in the service management tool to obtain the candidates. In these implementations, technical teams had to create a new change (just as they would as any other change), but they select a particular type of a change (called a *standard change qualification*). This type of a change request will come ordained with a template for capturing all the necessary details. Once they hit Submit, a change ticket number is generated, and the change ticket's workflow is designed to move the change ticket to the change management team's queue.

Step 2: Screen the Candidates

The change management team typically receives a number of candidates for standard changes. In fact, it is my experience that technical teams dump their entire list of changes in the qualification list for standard changes because they typically wouldn't prefer to work with normal changes, and standard changes make their life a lot easier.

The change management team vets the changes and the justifications and cross-checks from their records the number of times certain changes have been done in the past and their outcomes. If a simple change with low impact and risk was screwed up once before, the change management team would probably reject such a standard change candidate up until the point that such changes are implemented successfully at least a few consecutive times.

The vetting generally includes plenty of phone calls and conference calls with the technical teams to understand the change (if needed), and extensive analysis is done. The intent is to ensure that only good changes get represented in the SCAB, and the chaff gets cleaned out in this step.

Step 3: SCAB to Qualify Standard Changes

The SCAB is a virtual committee that I have conceived of (you will not find it in ITIL publication or elsewhere). It consists of the set of (wise) people who get to decide which change candidates can be standardized and which should remain as normal. The board members are picked to cover all areas of IT to provide a full 360-degree dimension for the decision-making. I would probably have the heads of infrastructure, the cloud, networks, applications, and databases represented on the SCAB. This will provide good coverage across IT.

The SCAB is built along the same lines as the CAB. The CAB is more dynamic in nature with its members differing for the specific changes that get presented. However, the design of SCAB is to have a standard (static) set of members who collectively make decisions about standardizing changes.

A representative from the technical team will be asked to present the change, explain justifications for why the change can be standardized, and answer all questions posed by the SCAB satisfactorily to have a change standardized. In my experience, at times, the SCAB just looks at a certain change and says, "Yeah, that's a standard change." The team's representative, although present, will be happy to get clearance with minimum work.

Step 4: Develop Change Models for Standard Changes

When the standard changes are approved by the SCAB, the change management team gets to work developing change models for the changes that are standardized.

A change model is a way of predefining the steps that should be taken to handle a particular type of change in an agreed upon way, according to the ITIL service transition publication.

In this step, a standard change template is created for a particular change (say running batch jobs), and workflows are created for, say, activity 1 to log the details of the batch, activity 2 to vet, activity 3 to program the batches, activity 4 to run the batches (automatically), and activity 5 to verify.

The activities defined for a batch job run are much different compared to a database re-indexing. A different set of activities is loaded in this workflow.

So, for every single standard change that's approved, the change management team has to develop change models individually, which is time-consuming. But, it's a one-time activity. Do it once, run it a zillion times!

With this step, the process for standardizing changes ends. Steps 5 and 6 are independent activities that can be carried out any time after changes have been standardized.

Step 5: Implement Standard Changes

The change management team informs the technical teams that the standardized change models are available on the change module of the service management tool for their perusal. The next time, when the situation arises to implement a change that is already standardized, the technical team creates a change ticket based on the standard change model for that particular change. The workflows are laid out beautifully for them to follow and carry out their change. An important thing to note is that the technical team can carry out a standard change only when the circumstance (trigger) presents itself.

For example, the SCAB while standardizing changes can impose constraints such as "For this particular standard change involving file backup, it can be done only during the following hours: 0000hrs to 0400 hrs." So, the technical team must comply and carry out the standard change only during the window. This is just one example of a constraint; there could be many others that the SCAB could impose.

All standard changes must be closed, and the closure of the standard change is the final task in the workflow. This is one of the areas where I find most teams lacking. Teams are good creating changes and carrying out workflow tasks until they hit the implementation stage. Once a change is implemented, they forget all about the closure as they get consumed in other technical stuff.

Step 6: Monitoring and Auditing of Standard Changes

Some critics of standard changes feel that standard changes are free tickets to technical teams to do as they please. They can do anything and everything that they please.

It is true that the technical teams can do anything and everything they please because they have all the accesses they need. But why restrict this criticism to standard changes alone? Teams can make changes even without a change in place (referred to as *unauthorized changes*).

So, how do we make sure technical teams don't abuse their power?

There are multiple ways of doing this. In one of my implementations, I leveraged the monitoring tools and the CMDB with some automation conjoining the two to identify unauthorized changes. The monitoring tools keep an eye on the components and services that come under its scope and on a regular basis compare the monitoring elements with the CMDB's classes and attributes. The tools do a cross-sweep across the changes registered in the system, with a similar change window as the anomalies are identified. If a change ticket is present and is in the correct state with all the requisite approvals (for a normal change), then the anomaly is closed down. Otherwise, an anomaly flag is raised, and the change management team is alerted. The change management team will further work with the teams to identify the problem and work through it. Once done, they close the alert to logically close the flag.

This change monitoring solution was extremely effective in combating unauthorized changes more than identifying noncompliance on the standard changes. Therefore, the second part of this step is to conduct logical audits on the standard changes.

I recommend a monthly audit schedule by the change management team to audit the standard changes. Or you might do it more frequently depending on the number of standard changes. If there are provisions or mandates to have an independent auditor perform audits on standard changes, then they normally do it either once every six months or once a year.

CHAPTER 9 MANAGING CHANGES IN A DEVOPS PROJECT

Auditing standard changes is simple enough. Pull all the standard changes implemented during a period. Identify a sample set. Note the constraints, triggers, and documented steps for implementing standard changes. On the standard change tickets, check whether they meet all the requirements. There are a number of audit tools available, or a simple Excel spreadsheet will do. The audit process is a separate process by itself, but for standard changes and in the context of DevOps, it can be simplified to include only the items that matter.

CHAPTER 10

Release Management in DevOps

We have come to the end of the adaptation transformation of ITIL processes into DevOps projects. Release management is the final (major) process that needs to be adapted to the swift and Agile way of developing and promoting deployments into various environments.

The release management process is an inherent part of both service management and software development. It has stood the test of time over the years and has changed rapidly as the technologies have evolved and the practices have transformed. It is time that release management is adapted to deal with an influx of automation, the flair of collaboration, and the management abilities needed to see releases through.

As always, I will jump into the ITIL release (and deployment) management process and provide an overview of the process for managing releases. I will follow it up with the DevOps way of understanding release management and propose a DevOps adaptation of the ITIL release (and deployment) management process.

Change Management vs. Release Management

Change management is a process that is responsible for obtaining all approvals and authorizations from relevant stakeholders and to control what changes go in. The *release management* process, on the other hand, deals with the management of technicalities of the changes. For example, let's say that a change has been raised to deploy a new version of the software. The change management process exists to put the change in front of the jury (CAB) and to help obtain approvals and authorizations so that the change can be deployed seamlessly. The release management process manages the requirement gathering, coding, testing, and deployment activities of this change.

The two processes work together, exchanging information from related activities in the processes. This is illustrated in Figure 10-1.

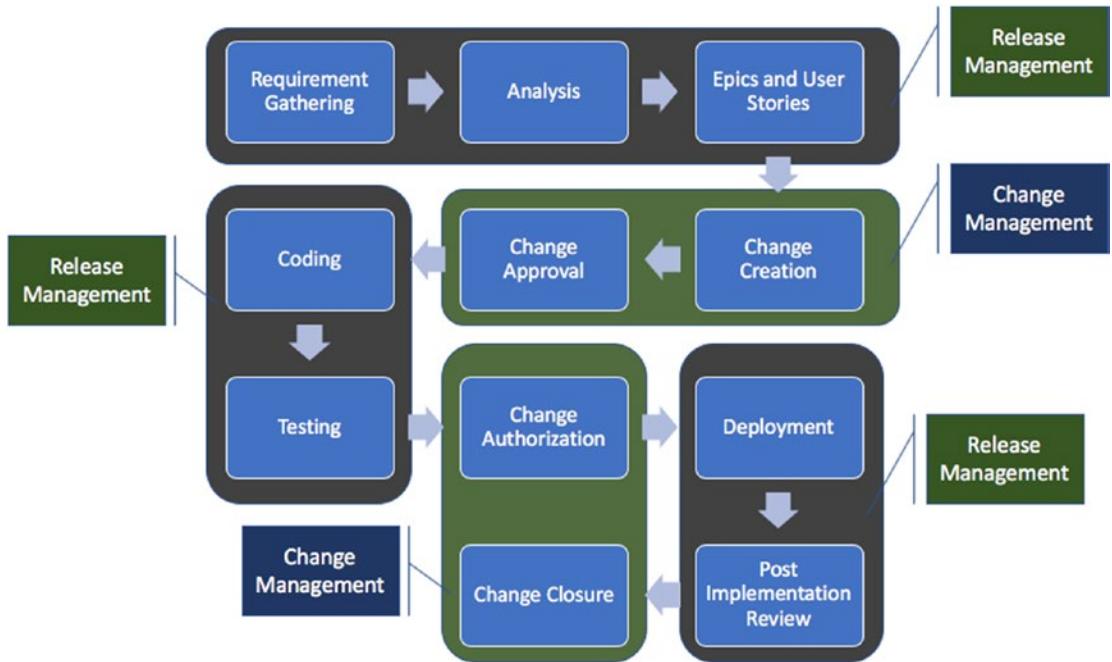


Figure 10-1. Illustration of the overlap between the change and release management processes

In Figure 10-1, setting the scope, creating a plan, and gathering the requirements all come under the auspices of release management. Once there is a plan in place, including the known dependencies and impacts, the change management people do their bit of formal vetting and approval, before giving the go-ahead to begin development.

The change management approval process is necessary before the development begins because of two reasons.

- It ensures the efforts that go into development and testing are not wasted if change management decides not to approve the changes.
- If there are modifications to the solution proposed by the CAB, then rework can be avoided.

The release management process manages the overall planning, build, test, and deployment portions of the project. However, after the tests have concluded, the release status is put back into change management's court. Change management's authorization is a necessary step to ensure that all the entry criteria for deployment to begin have been met. It also provides oversight before changes to production are set to begin.

The deployment and post-implementation review activities are owned by the release management process, and the results are duly reported to change management for change closures with the appropriate status.

In this sample process, the release has been ping-ponged between release and change management processes at least six times. It can go higher; the higher the number of exchanges, the better it is for ensuring the quality of governance and sharing accountabilities.

It is also true that some of the activities that originally showed up during the change management process (such as deployment and post-implementation review) actually belong in the release management process. This was intentionally done to ensure the continuity of defining the change management process.

Release Management vs. Release and Deployment Management

In ITIL v3, the process pertaining to releases is called *release and deployment management*. There is no release management process in ITIL today. In the software development lifecycle, we use the term *release management*, and a process is associated with it. So, the question to ask is, what is the difference between release management and release and deployment management processes?

There is absolutely no difference between the two processes. The ITIL process looks at a process from a service provider's perspective, and the software development's process gives it a development team's touch. So, whenever I refer to release management in this chapter and book, I am also referring to the release and deployment management process. Both are one and the same.

In fact, in the previous ITIL version, ITIL v2, the release and deployment management process was referred to as the *release management process*. Only in v3 (2007) did the process get coined the release and deployment management process.

Basics of a Release

The release management process is a vast process that includes everything from requirements gathering to the nuances of planning, building, testing, and deploying. The entire project gets played out within the realm of the release. Therefore, it is key that the process is understood, precisely planned, and executed with razor-sharp precision.

In this section, I introduce certain fundamentals of the release management process that are common across the ITIL and software development areas.

Release Unit

A *release unit* is the combined set of items (configuration items, software files, and so on) that are released together into the live environment to implement a certain change ticket or multiple change tickets. The crux of the release unit lies in the grouping that gets deployed as one unit.

Figure 10-2 shows an example of a release unit.

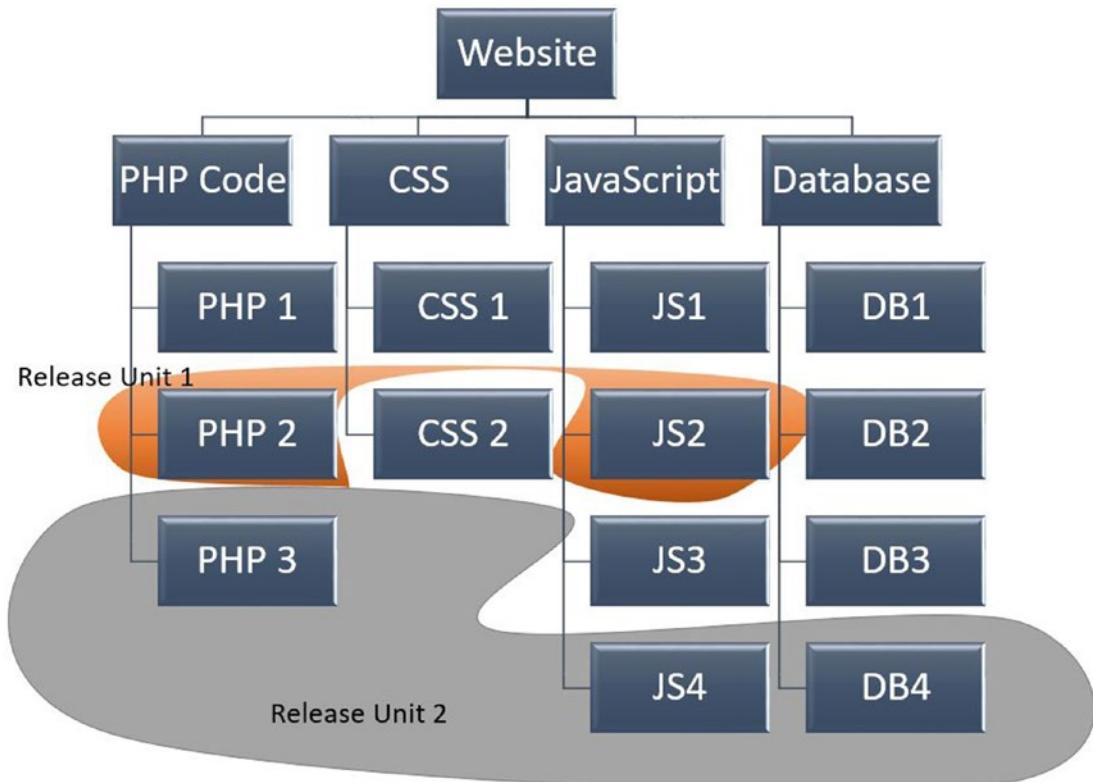


Figure 10-2. Release units for a web site

Figure 10-2 indicates the various components of a web site that is under the scope of release management. In this web site, there are PHP, CSS, JavaScript, and database components. Under PHP, there are three files that make up the PHP code for the web site. Likewise, for CSS, there are two files, and JavaScript has four files. Also, there are four databases storing the web site's data.

In this illustration, I have highlighted some boxes as release unit 1 and release unit 2. A particular functionality is being delivered by making changes to the files PHP2 and JS2, and together, the grouping is referred to as a *release unit*. This grouping is because the functionality is delivered together from both the files. Without the JS2 file, the PHP2 file alone cannot deliver the required changes. Likewise, the JS2 file cannot do the job alone. Therefore, the files PHP2 and JS2 are grouped together as a release unit.

The next release unit, release unit 2, is a bit more complex at the outset. It requires changes to the files PHP3 and JS4 and to the database DB4 as well. Just as in release unit 1, all three need to be deployed together to be able to deliver the required changes. Therefore, they are grouped as a release unit, united by a common release unit number, release unit 2.

Release Package

A *release package* is a combination of one or more release units that will be deployed together as a single release. In Figure 10-2, let's say release units 1 and 2 are slated to be released under a single umbrella. This would constitute a release package.

A release package is not merely a grouping of release units to be released in one window. The release package during its planning stages has to consider the effects of certain release units being together, especially if there is a strong dependency and high business impact between each other. The next steps for a release package planning are to ensure that sufficient resources are available to build, test, and deploy the release and to see whether sufficient infrastructure resources are available for accepting the new release packages. There could be many such considerations that decide how you will plan and execute a release package.

There is more to a release package than the release units itself. The associated deliverables such as documentation, training, and compliance set also make up the release package.

It is also important to note that release packages are uniquely identified with release numbers. The decisions for numbering are documented in a release policy, which states the manner in which release numbering takes place and also sets the rules for other release-related decisions.

Types of Release

The release policy defines the types of releases in an organization and at a project level. The distinction for differing types of releases is based on the complexity of the release, the schedule it maintains, and its associated business impact. Some organizations may decide to do a release every week with some minor updates and then at the end of the month may plan for some serious packages to make its way into production. The release policy can state what constitutes a major release, a minor release, and even types of releases. I know of an organization that used to have medium and urgent releases. This may not find its way into ITIL, but nonetheless, if the organization finds it relevant, then that's all the more reason to keep it and define a number of types as necessary.

According to the ITIL publication, there are three types of releases, covered next.

Major Releases

Major upgrades to software generally come under *major releases*. The business impact of major releases can be anywhere between high and critical. This type of a release is the mother of all releases and takes priority if there is going to be a minor release happening at about the same time. A number of resources are usually dedicated to the building and executing of a release, and from a compliance angle, all the hawks should watch it with extra attention.

In my experience, major releases are far and few between. In most cases, they are done on an ad hoc basis, with some organizations deploying at least four major releases in a year. For example, you might notice the updates being applied to the Windows operating system. Some changes are quick and may not even demand a restart. However, additions, modifications, or the removal of integral features happen on the back of major releases that could require several minutes of installation followed by multiple restarts.

Minor Releases

As the word *minor* suggests, *minor releases* include release units that are small and do not usually bring down the business if the release goes to go south.

Minor releases are usually carried out as often on a weekly basis or as late as monthly. It all depends on the number of changes that are getting pumped in and the number of resources available to work on them.

Emergency Releases

Emergency releases are the planning and execution counterparts of the emergency changes. They come into play on the back of an emergency change and are deployed (usually) to fix an incident and to avert negative business impact.

The number of emergency releases reflects negatively on the organization and the project. Therefore, this is a type of release that's not preferred or planned but rather gets imposed by the turn of events. It is also not uncommon that the release policy allows emergency changes to be helicoptered in only between releases, say between two minor releases.

Early Life Support

Early life support is one of the key lifecycle stages in the ITIL service transition phase. In non-DevOps projects, the team that supports the services is different from the team that builds it. The gap that exists between the two teams is bridged through a temporary phase of overlap right after the deployment is done.

The build team provides support right after deployment, and the support team usually supports them. So, when incidents come about, the build team is ready with the fixes right away because they know the product in detail. At the same time, the support team that's monitoring them will get a chance to learn. Since the support is provided during the first few weeks after deployment, it is referred to as *early life support*. Some organizations refer to it as a *hyper care period* where the best resources are tasked with providing support during the product's initial days.

Generally speaking, during early life support, the SLA does not kick in nor does any of the other parameters that are used for measuring the level of service.

Early life support is completely eliminated in DevOps projects as the team that builds and the team that supports are one and the same, and therefore having a hyper care period isn't relevant. Or in other words, the product that's being serviced is always in hyper care support, meaning that it gets the best support possible at all times of its lifecycle.

Deployment Options

When the software is ready to be deployed, we can deploy it directly across all the target devices in parallel or do it in pieces. Based on these strategies, the ITIL service transition publication talks about two basic types of deployment options.

1. Big Bang option
2. Phased approach

Figure 10-3 illustrates both approaches.

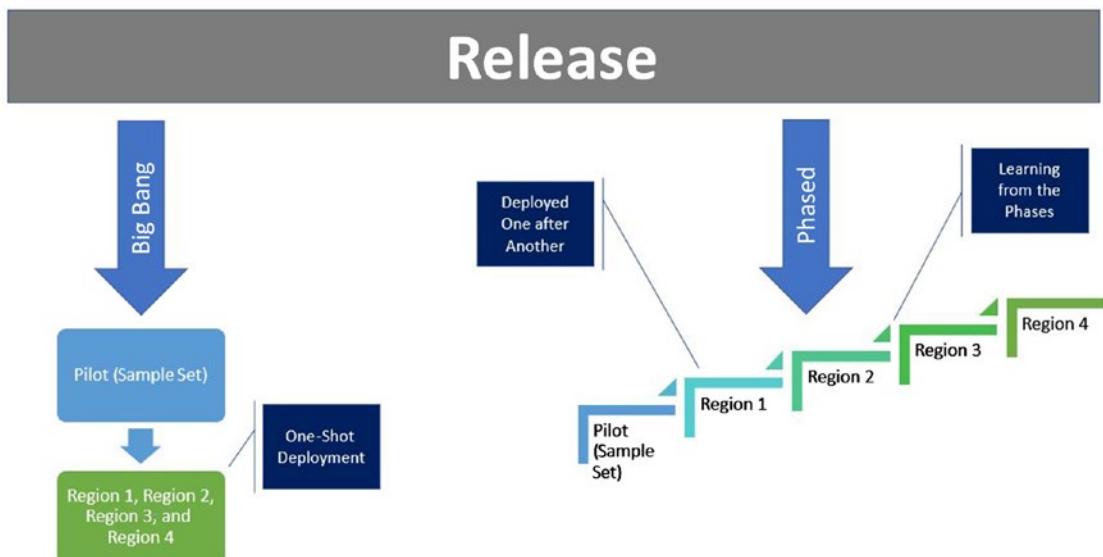


Figure 10-3. Big Bang and phased deployment options

Big Bang Option

The Big Bang option is derived from the Big Bang theory, which states how the universe came into being from a single super force. Likewise, when software is deployed, it gets deployed to everything that's under the scope at the same moment. In other words, all users will get to experience the software (or the trauma of deployment) at the same time. This type of deployment is referred to as *Big Bang deployment*. It is also called *parallel deployment*. In Figure 10-2, after a pilot deployment to a sample set of users, the release package gets deployed to regions 1, 2, 3, and 4 at the same time.

The upside is that all users will be in a position to enjoy the upgraded services at the same time, and the service provider can claim to be consistent with its services. This is generally preceded by a pilot (or multiple pilots) to ensure that the software does work. When the pilot is deemed successful, a time is set, and the users are made aware, and then the entire scope of the targeted system will receive the release package.

The Big Bang option is almost never considered in these modern times. You might have noticed that releases happen to certain smaller regions first (pilot) followed by, say, the United States. iOS updates are known to follow this pattern. The downside of deploying to everyone at once is pretty significant. Any screw-up will result in a disaster, and the negative business impact that follows is unbearable. Therefore, no organization likes to take chances by pushing everything out worldwide in one go.

Phased Approach

The alternative to the Big Bang option is to deploy in a phased manner. In Figure 10-3, an initial pilot to a sample set of users is followed by a phased deployment to region 1 first, followed by regions 2, 3, and 4. It is okay for the deployments to have weeks and months between them to allow for the learning to sink in and corrective actions to be implemented before the following release. This is the biggest advantage of a phased approach. Organizations can manage their risks and target their audience based on various parameters. For example, say an organization would like to deploy packages during usual downtimes in different regions of the world. This may be the Diwali season in India, Christmas in the United Kingdom, and Rosh Hashanah in Israel.

I don't see any obvious downsides to the phased approach except that it requires a lot of continued planning and differences in release version between users, so this may end up being a support challenge. But there are a number of ways to mitigate this.

There are multiple variations of phased approaches that can be conceived apart from the geographical deployments described earlier.

- Different features are deployed separately, so users can enjoy certain features first and then the rest get them later.
- All users face downtime at the same time, although the deployment happens in phases. This usually refers to deployments taking place right on the tail of a previous one.
- A combination of geographical deployments, feature-wise deployments, and downtime for all.

Four Phases of Release Management

Release and deployment management has four major activities, or *phases*.

1. Release and deployment planning
2. Release build and test
3. Deployment
4. Review and close

Release and Deployment Planning

A good amount of planning has to go into release activities. A good plan is half the job. To ensure that the release is successful, it is critical that the architects and other experts brainstorm various possibilities, risks, and mitigations.

Before the plan gets underway, change management typically provides approval to start the planning process. However, in practice, the approval to create release and deployment plans is provided from a different body such as a transition management group or a group that governs the projects that are chartered. In principle, these bodies govern the changes made to the system, and this can be the equivalent of change management authorization to create release and deployment management plans (part of the transition plan in the transition planning and support process).

Release Build and Test

The release and deployment plans are submitted to the CAB. The plans are dissected from every possible angle to identify loopholes and vulnerabilities. Upon successfully passing the CAB and change management scrutiny, the authorization to build and test the change is provided.

Building a change amounts to developing the code, getting hardware ready, or addressing the prerequisites for building the change.

There are various types of testing. The most popular ones are unit tests (UTs), where individual components of a change are tested in isolation. Upon successful testing, the individual components are conjoined, and a system integration test (SIT) is performed.

After a test successfully passes, users are asked to test the function in the user's environment to check whether the change meets the requirements that are needed. This is called *user acceptance testing* (UAT). The testing is deemed complete after the user provides the okay that all elements of the change meet the requirement and are good to proceed.

The definitive media library (DML) is a repository where all the original code, software licenses, and other software components are stored, physically and logically.

When release and deployment management provides ample proof that the testing has been successful, change management provides authorization to store the code/software in the DML (discussed in Chapter 6).

Deployment

The results of release and deployment testing are brought before the CAB once again, and the results are vetted for possible complications and unseen bugs. When the CAB and change management team are happy with what they see, they authorize the change for deployment during the planned change window.

Deployment is a common term for implementation. It could include retiring services or transferring services to another service provider as well. For simplicity, I'll just refer to it as deployment.

Deploying release packages is a specialized skill and calls for the alignment of a number of parameters. There are a number of approaches to the release package. The Big Bang approach is used when all the CIs are targeted to receive the package at the same time. Say there are 10,000 workstations that need to get a security patch. All 10,000 systems will receive the release package during the same window.

This method is rarely employed as it has the ability to choke the network. And, if there are any mishaps, all the targeted systems could be affected, causing severe damage to the customer. The most popular approach is a phased approach, where the release is staggered through multiple phases to minimize complications and avoid network choke. In the same example involving 10,000 systems, it could be phased to target 1,000 systems a day and to run the entire release cycle for ten days.

Review and Close

After deployment, the release and deployment management process conducts a review (post-implementation review) to check the performance of the release and assess the targets achieved. Lessons learned are documented in the KMDB. The release is closed after the review.

Releases in DevOps

The release management phase we looked at earlier in this chapter is highly sequential in nature. Its sequential nature is the backbone of the waterfall model of project management. Agile is iterative in nature, and so is DevOps. So, can the sequential release management process be compatible with the iterative DevOps model?

Yes! Release management is sequential in nature. It waits for one activity to complete before the next kicks in. I am referring to some of the high-level activities such as planning and deployment. Unless planning is complete (consider a road map), development cannot begin. Unless testing is complete, it should not be deployed. Therefore, it is fair to assume that the sequential nature of release management is best handled the way it is designed. But it is also possible to make it a whole lot more Agile and give it a hint of “go-with-the-flow” flavor.

Sequential and Iterative Nature of the Process

Figure 10-4 illustrates the release management process’s iterative and sequential phases.

The release and deployment planning was considered to be a detailed exercise that spelled out all nuts and bolts of the release, including the date and owner. With the advent of Agile, the planning bit of the exercise was simplified, with more importance given to the subset of requirements in development. In addition, a road map was created

by the strategists in the organization. The planning exercise is an iterative process where the immediate bits of the puzzle are figured out, and once it is close to being delivered, the next bit is brainstormed and planned. This way, the planned items more often meet the deadlines assigned to them and make sense, instead of a whole bunch of missed timelines and detailed explanations and root causes for the misses. In effect, the planning phase that used to highly sequential is now carried out in a highly iterative manner.

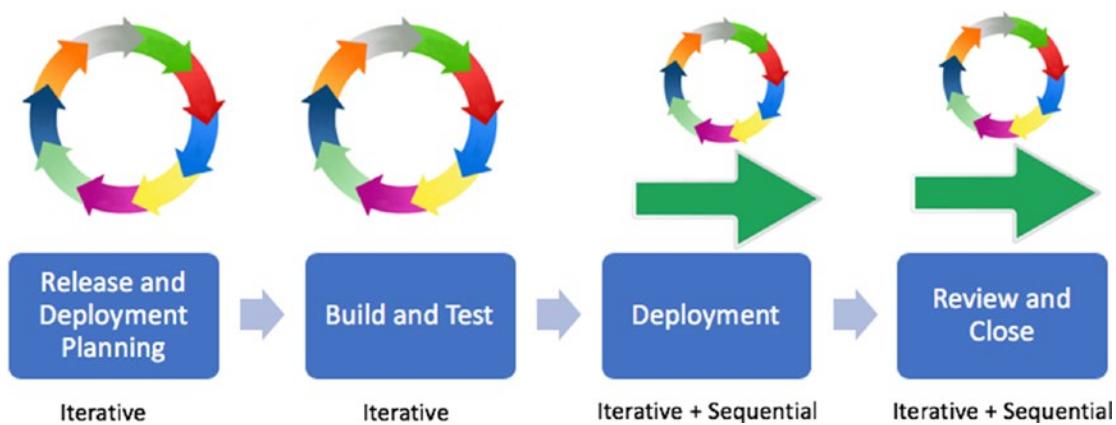


Figure 10-4. Release management iterative and sequential components

The next phase consists of the build and test activities. These activities are technical in nature, and in Chapter 1, I explained how the continuous delivery phase is built, which is iterative as well.

When the product is built and tested, it is time to deploy it. In continuous delivery, the objective is to keep the package deployable at all times. Therefore, when we implement continuous delivery, the deployment phase becomes sequential as the build and test activities need to be complete, and a suitable window needs to be identified for the deployment to happen.

Let's say we are going in with a continuous deployment type of delivery, where every time a piece of code gets checked in, it gets tested automatically, and if it meets all the set criteria, it gets deployed automatically. In such instances, the deployment piece does not wait for a pre-condition but rather goes with the flow (the flow being the test criteria being met and the coding and testing taking place in iterations). So, the deployment phase too becomes iterative in nature.

Finally, review and close follow the previous deployment phase. If the deployment phase is sequential, the review and closure to take on a sequential tone (and likewise for an iterative nature).

Release Management Process Adaption with Iterations

In DevOps, release management has not been completely transformed. It has become stronger through iterations. The activities in release management are now viewed by a different pair of lens that's ready to accept the facts based on the things at hand, rather than foretelling the future.

Using Agile Release Trains

The range of planning that we have started to do under releases is not limited to a sprint alone, which is the Agile/Scrum way of working. However, using the SAFe framework and applying release management to Agile release trains (ART) gives us a steady plan for the upcoming ten to twelve weeks. The entire ART represents a release with software packages pouring in every couple of weeks at the end of each sprint. When we put the sprints together along with their outcomes, the end product is the release package.

Applying Release Management to Continuous Deployment

In the DevOps world, the release management process can be adapted depending on the kind of process (continuous delivery or continuous deployment) we leverage. Let's say that we plan to employ continuous deployment where every time a package is tested successfully, it gets deployed automatically. Release management's role here is to ensure that the path to production is stable, relevant, and consistent.

The release management process will have a lot of planning and execution during the initial two phases rather than the final two phases. Still, if a faulty package makes its way to the production, the ball falls back into the release management's court to fix the pipeline and the associated factors that make the pipeline work (such as testing scenarios, scripts, and so on). Also, release management has to identify multiple release windows for the deployments to take place because the possibility of deployments happening multiple times a day is routine in a continuous deployment process.

Applying Release Management to Continuous Delivery

In continuous delivery, however, saneness can be maintained to a certain extent. The release management process becomes bimodal, with release planning and builds/tests using the iteration model and deployments and reviews taking the traditional sequential approach.

The plan for continuous delivery works well with ART with the planning exercise being done once every 12 weeks and refined as the sprints go along. The sprints are executed in iterations with the software packages, getting them to a state of readiness but not getting deployed. When all the pieces of the release are developed and integrated, the deployment happens (sequential) followed by a review of the release.

Most organizations will tend to go with this approach, mainly because it gives people in charge a sense of control. Since continuous delivery still commands a manual trigger before deployment, the decision-makers feel comfortable in opting for a process that not only accelerates production but also awaits a formal order before hitting production.

Maturity is leading the way toward continuous deployment. The decision-makers, after a few releases, will realize that their decisions have always been backed by the figures, that the releases put forth in front of them have always been good for production, and that their decisions have become just a formality. So, the end game will always be with continuous deployment.

Expectations from Release Management

Every process has to meet certain objectives to justify its existence. Incident management exists to reduce downtime and to restore service as soon as possible. Change management is implemented to control the production environment from malicious changes. Likewise, at a high level, release management exists to ensure that releases are deployed successfully in the targeted environment.

In DevOps, the processes don't get away easily with such simple objectives. There's always more to the story, such as productivity, effectiveness, efficiency, and automation.

The DevOps release management process's expectations are multifold. We are not happy with just successful deliveries; we need them faster, and the process for delivering must be consistent and not two-paced.

Since there is potential to make hundreds of changes in a day, it is critical for the release management process to provide strong auditing capabilities. Most important, we need to trace the feature changes back to their requirements in a straightforward manner.

Where there is DevOps, there's automation. Release management in DevOps must be automated to a greater extent than before is one of the key expectations from the process. Automation ensures a systematic execution of work processes, an avoidance of defects owing to human errors, and consistency in delivery.

Blue-Green Deployment

Seeking downtime for releases is a thing of the past. In DevOps projects, carrying out deployments without downtime is the norm, and release management must do this at a minimum. There are a number of ways the process could achieve this. One such example is the *blue-green deployment approach* where two environments are run in parallel (each of the environments is designated with the color blue or green).

Figure 10-5 illustrates blue-green deployment approach. We have two parallel environments, designated as blue and green. Both the environments are identical; however, one of them is active and the other passive. In this example, let's say that the blue environment is active and the green is passive. This refers to the load balancer routing all the user requests only to the blue environment and not to the green environment.

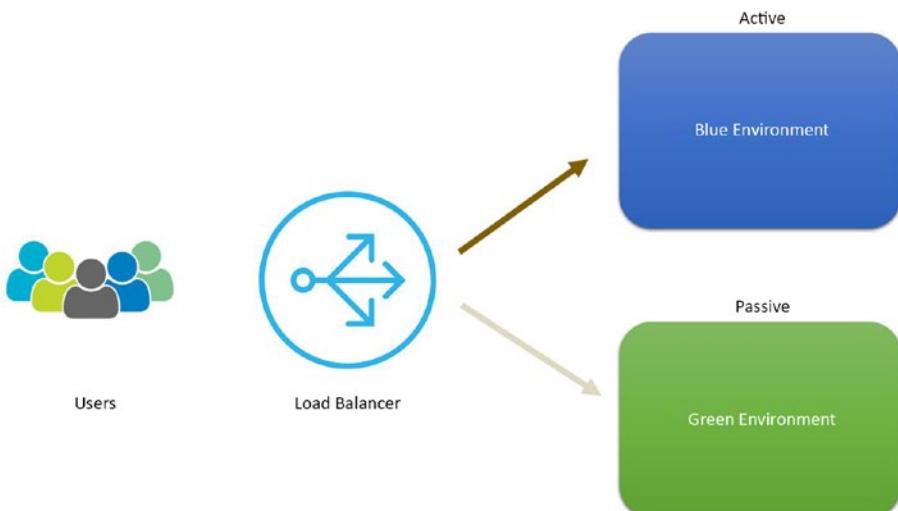


Figure 10-5. *Blue-green deployment*

Let's say that we have a release on hand that requires mandatory downtime to install and configure packages, followed by elongated sanity reviews. In this scenario, the passive node (green) gets deployed first. As there are no user requests getting routed, there is no question of seeking downtime. The users continue to operate normally in the blue environment. When the deployment is successful and the environment is production ready, the load balancer routes all the user requests to the green environment. While users are busy operating in the green environment, the blue environment goes down, gets deployed, and becomes production ready. The load balancer can either be set back to blue or be shared between green and blue; this is a decision of the architects. *Voila!* Both the environments have been deployed with packages that required downtime, but the users never felt the effects of the downtime.

The Scope of Release Management

The scope of release management based on the activities we discussed falls across the software development and software support teams. Because of the overlap, traditionally the process has lacked ownership where one entity throws it back to the other stating various reasons for complications. For example, during the time of deployment into production, where production is owned by the support team, the development team takes their hands off the deployment and related configuration settings. This will, in turn, become a support issue. But the support team is rather clueless, and they need the help of the development team to set the right configurations. The only way out is that the two teams work together to move past the obstacles and to make the release a success.

Enter DevOps! The reason why DevOps adapts so easily with the release management process is that it has already solved the unending problem involving collaboration and cooperation between the development and support teams. Under the DevOps umbrella, both the teams are placed on the same team, and they don't have a choice but to work as a team. Either both fail or both succeed. The obvious sane choice would be to cooperate and see through the deployment and the acceptance parameters of the release.

As per the Agile and DevOps principles, the entire ownership of the release will lie with the team, as they are self-supervised, although we have a formal release manager still in existence to give the release a touch of what the customer is thinking. With the entire team working toward the release's success, the argument about whether development or support is taking the ownership of the release is put to rest.

Automation of Release Management

The release management process as defined in ITIL is solid and mature. I don't see any need for it to be altered or adapted for a DevOps project (in terms of the process activities). However, what we can still do is make it more efficient by using the power of technology to automate certain aspects of release management.

Of the four phases of release management, at least two of the phases can be automated: release build and test, and deployment. The majority of the time of a release is taken up during these two phases, and it absolutely hits a home run with the automation running against activities that are designed to eat up most of the release time. Another important concept in deployment is a rollback plan. This plan is used when a deployment fails and needs to be recovered and restored to the previous state. Automatic rollback is achievable using tools such as Ansible and Puppet, and it can help to restore the original build and configuration efficiently.

The automation for releases is done on the pipelines that are built on release orchestration tools such as Jenkins and Urban Code. I touched upon the pipelines in the initial chapters that form the basis for building and implementing continuous delivery and continuous deployment. The pipelines define the various stages of software development and testing that make the software acceptable to end users. The pipeline further makes sure the developers and testers spend their efforts in doing what they do best---coding and writing scripts. The execution of these activities such as software build and testing is taken care of by the orchestration tools. The results are duly updated, and the defects are logged in the designated project management tool.

To state an example of the kind of efforts that can be saved through automation, consider this: every time you make a change to a functionality, there is a need for the regression issues to be tested. Testers spend a good amount of effort testing for regressive defects every time the changes are put back into the development and testing environments. The time taken to test regression issues forms a major portion of the overall testing efforts. We can automate regression testing, and every time a new check-in happens, the testers don't have to fret anymore. The machines take over automatically by checking all scenarios of regression and reporting on the status. If there are any defects, the orchestration tools generally have the privileges to log defects automatically based on the result of the regression run. Just merely logging defects saves testers tons of time that they can use to write more scenarios and automation scripts to make the testing process more accurate and intensive. Before the time of automation, all the efforts going toward test execution and defect logging were wastages that were effectively discarded.

The automation of release management activities is not restricted to testing alone. The version control system that is part of comprehensive configuration management (CCM) is a major catalyst for making automation work smoothly. Imagine working on automation without a single source of truth. It's like an airplane trying to land on a runway with the runway lights out. There is no reference point for the plane to land, and similarly, the coding, testing, and automation activities designed around the activity fail without the integration of CCM and automation.

The DevOps Release Management Team

I made it pretty clear that the ITIL release management process is good as is with some iterations plugged in. The process is good; however, the people who run the process in a typical service management project may not be as lucky. The release management teams that are responsible for driving the process from the planning sessions until the closure of releases are no longer needed. Am I crazy to suggest that we have the process but not the people who run it?

Remember that the objective of DevOps is to ensure that productivity increases and that no humans are needed for repetitive activities. The release management process falls into the direct firing line of the DevOps objectives, and this has consumed the team that runs the process. Yes, the reasons are obvious to be with: the majority (or all) of the build, test, and deployment activities are done by machines. These two activities account for more than 70 percent of the overall release management activities. So, automation has simply killed the release management team! Yes, that is indeed true, but what about the remaining phases like planning, reviews, and closure? You certainly can't ask the machines to do these too. Definitely not---at least not the planning phase because of the human's cognitive powers, which cannot be matched by the machines (yet).

Then who is going to carry out the human part of the release management process? We will explore this further in the next section.

Before we embark on specific roles, we should be wary that DevOps teams are made up of two sets of broad roles that people play. There are developers, and then there are operations. The way they communicate or the terminologies they use is not even the same. When an operations person talks about a service, that person basically means the service we are offering. The same service is also a product that is being developed. But the developer might not call it a service but rather refer to it as software or a system.

That's not the only difference. Even the tools they typically use are much different from one another. The development team uses a tool such as Jira for managing its product backlog, while the operations team uses a tool such as ServiceNow. So, the lack of a single source of truth might lead to more differences.

This is precisely the reason why release management is handy and provides the common understanding between the two teams. The teams are able to understand each other because they stretch across both sets of roles and people. This helps the overall DevOps teamwork and collaborate for the betterment of the project.

Release Management Team Structure

In DevOps projects, having a release management team is a luxury, given that the entire focus is on optimization and there are fewer human hands to work on repetitive actions. However, in the industry, many DevOps projects still leverage release management teams to make releases work. The structure, however, differs from one organization to another. Let's explore some common ones.

Separate Release Management Team

This is the traditional approach where a separate release management process exists to take care of releases. The release management team is generally governed by a separate release management practice that ensures that common release management practices are followed across all projects under their control. So, in a way, there is an amount of standardization they bring to the table.

Having additional hands to work on releases is a godsend, especially if the release management teams start working with the team rather than acting as stage gatekeepers. However, since the release management team gets helicoptered into projects, their intimacy with the ground situation, solution, and customer expectations can be lacking in some ways. Also, this option costs more and affects the cost of delivery.

Release Management by the Delivery Team

The delivery teams (development) can run the management of releases. This is helpful because of their closeness to the development and their intimacy with the requirements and customers. Organizations that cannot afford to invest in separate teams go for this approach.

On the downside, such a release management team could end up playing both sides, development and release management. Unfortunately, the conflicts of interest will be written all over the decisions made. This option is generally implemented only when there are limited finances, and I do not recommend it.

Release Management by the Operations Team

In contrast, we could have the operations team working on the releases. They are close to the production and the users, so they make a good choice for managing releases, right?

Not really. Like the separate release management team, they end up being on the outside of the solution and could end up not knowing what the customer wants. But wait---in a DevOps team, the operations team works closely with the development team. So, they are close to the solution. Yes, that is true, and therefore they make the second best choice for a release management team. You might be thinking about the best choice, though. I will introduce the best option for a release management team in the next section.

Welcome, Release Manager, the Role for All Seasons

You can probably guess that a release manager is somebody who manages releases! This is a simplistic definition of a release manager and does not emphasize the value of the position. Yes, in an ideal sense, the release manager is in charge of managing releases and ensures that the releases flow as per the design with no anomalies. The release manager also makes judgment calls on the contents of a release, sets deployment parameters, and manages stakeholders.

However, there is more to a release manager than this definition. Remember the picture from Chapter 1 where I illustrated the actual distance between the development and operations teams in a traditional project? I brought that distance to life with a picture; see Figure 10-6.

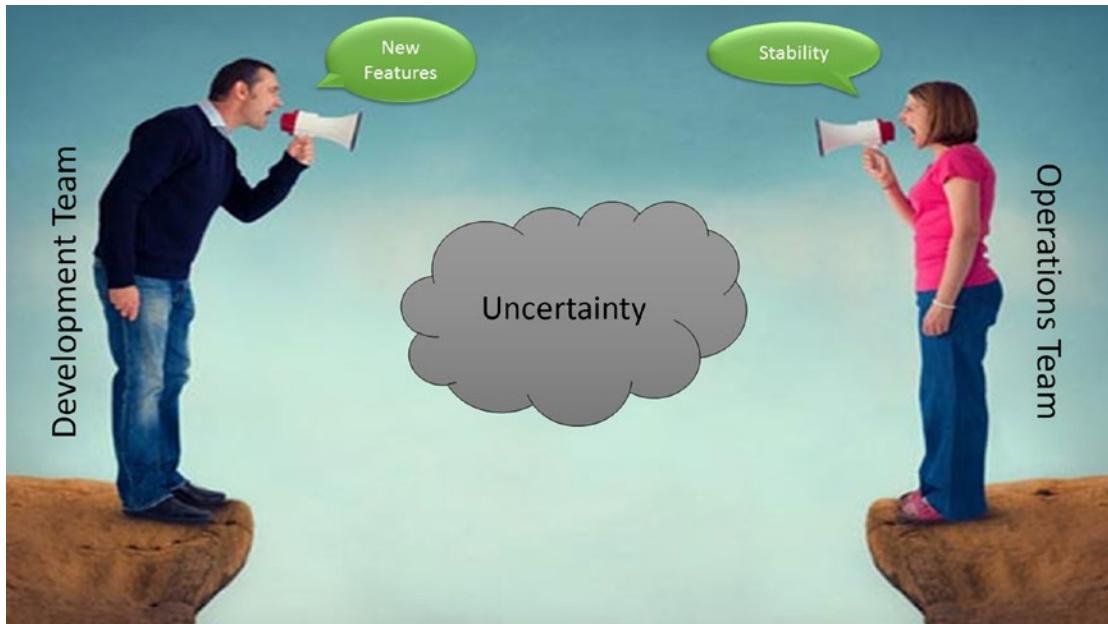


Figure 10-6. Cloud of uncertainty in a traditional project

I also wrote in Chapter 1 that through the merger of the development and operations teams, we are breaking down the barriers, and the teams don't have to be on either end of the cliff. Rather, they can come together and work as one. This is the premise for making DevOps work.

However, how do you bring the two teams together? Where's the bridge? See Figure 10-7.



Figure 10-7. Bridge between development and operations teams

The bridge between the teams is the release manager. Release management's scope starts with the business analyst, runs through the development team, and ends with the operations team. So, in effect, release management cuts through multiple teams and brings all the involved teams to the table to get its job done. The person who manages the releases, the release manager, is at the forefront of releases---understanding requirements, planning for its execution, running the code builds and tests, deploying them onto target locations, and finally carrying out the checks and balances to put a logical closure to the release. Throughout the lifecycle of release management, the release manager will have brought multiple teams to the same table, challenged the teams to work together to achieve what is needed, and built a bridge between the development and operations teams, as illustrated in Figure 10-7. When there is a bridge, there is a certainty because the teams don't have to yell to make themselves heard but rather can cross over to the other side and work shoulder to shoulder to ease the workload.

The bridge between the teams would not be possible if not for the release manager. Yes, the management bit is what the person does as a hygiene factor, but building the bridge will actually get the job done with minimal controversies, conflicts, and escalations. There is nobody else on the DevOps team whose influence has such a wide span across the project. Take, for example, the Scrum master. The Scrum master is merely interested in the development side of things and ensures that the software packages are delivered for deployment on time and in an effective manner. Whether the operations team understood what they had to do to conduct sanity checks or how the operations teams will support the product in the future are none of the Scrum master's concerns, so that person is not a good fit to be a bridge!

Product Owners Are the New Release Managers

The release management team has been made partially redundant by machines. It is not absolute because of two reasons.

- You need an owner for the entire release management process that cuts across both development and operations.
- Cognitive abilities are very much in demand to ensure that the release management process succeeds and aligns with the objectives set forth.

The person who manages the entire release from end to end is the release manager and is still necessary. However, the release management role went from being a full-time position to a part-time one (statistically speaking), mainly because of the diminished work (thanks to automation). Capable release managers are

1. Well aware of the customer landscape, the requirements, and to an extent the business priorities
2. Fully involved in the development and deployment processes
3. Understands operations and their acceptance criteria

The person who could do all this in the past was the product owner (PO), and thus that person is a favorite choice for a part-time release manager. POs are an adequate choice mainly because of their closeness to the business and to the development and operations teams. The person was like a bridge between the two entities and was expected to keep the boat going in the most turbulent conditions.

Product owner are indeed the best person to plan considering that they had a first-hand knowledge of the priorities of the user stories in the product backlog. This gave them a clear advantage to pick and choose what went into a release, considering that we were not operating in a continuous deployment mode. Clubbing user stories for a particular release is rather a hard task considering the dependencies that exist between them, and the PO can definitely make it look easy.

The PO is expected to be part of the sprint and carries out the role in terms of clarifying requirements. Combining this with the control over how the code flows and the quality that gets embedded gives this person the authority, flexibility, and freedom to spin the yarn the way he always wanted. On the flipside, however, the entire responsibility for the release and the customer expectations lie heavily with the PO, making this person seem like a single point of failure. This is why we need to have powerful and robust governance in place to support the product owner to become successful. I can't comment enough on the importance of a governance body to steer the decisions and directions undertaken. The PO is only human and could take a wrong step once a while. If the governance body is there to support the PO, the product owner is stronger, and the product and the release have no other alternative but to become a roaring success.

Index

A

Active monitoring, 168
Adaptation, 73–74
Agile model
 flat hierarchy, 121–122
 no project manager, 122
 predictability, 124
 product owner, 123
 scrum team, 121
 single team, 122–123
Agile project management, 73, 80, 124
 incidents, 177
 problems, 178
 sprint planning (*see* Sprint planning)
 sprints, 178–179
 user stories, 176–177
Agile release trains (ARTs), 179, 251, 284
Amazon Web Services (AWS), 32, 80
Ansible, 33
Apache Subversion, 32
Application management teams, 115–116
Application support (AS), 19
Architect (ARC), 19
Artifact repository, 161–162
Automation, release
 management, 288–289
Availability management, 91
Azure, 32

B

Bamboo, 33
Behavior-driven development (BDD), 34
Big Bang deployment, 278–279
Big-ticket conflicts
 batch sizes, 68
 configuration management, 70
 continuous deployment process, 71
 feedback cycle, 69
 release management process, 71
 sequential *vs.* concurrent, 68
 silo culture, 69–70
Binary management, 161–162
Blue-green deployment, 286–287
BMC Remedy, 242
Brainstorming, 206–207
Business capacity management, 92, 94
Business continuity management (BCM), 95
Business relationship management, 84

C

Capacity management
 business, 92–94
 component, 95
 service, 94

INDEX

- Centralized version control system (CVCS), [30, 159](#)
- Change advisory board (CAB), [16, 102, 235, 243–244, 248, 257–258, 260–261](#)
- Change management, [101](#)
- CAB, [235](#)
 - CMDB, [235](#)
 - configuration, [235](#)
 - emergency changes, [239](#)
 - normal changes, [238](#)
 - objectives, [236–237](#)
 - protocols, policies, and processes, [237](#)
 - vs.* release management, [271–273](#)
 - resources and capabilities, service assets, [232–233](#)
 - scope, [233–234](#)
 - stakeholders, [231, 235](#)
 - standard changes, [239–240](#)
 - unauthorized changes, [235–236](#)
- Chef, [33](#)
- Commercial off-the-shelf (COTS), [115, 158](#)
- Component capacity management, [95](#)
- Comprehensive configuration management (CCM), [153–154, 289](#)
- CMDB, [154–156](#)
 - SCR, [157–161](#)
- Configuration items (CIs), [136–137](#)
- Configuration management
- application deployment, [150](#)
 - automation, [151–152](#)
 - CMBD, [147](#)
 - database, [151](#)
 - decoding IaaS, [149](#)
 - DevOps, [147–148](#)
 - PaaS, [149–150](#)
 - team maintenance, [153](#)
- Configuration management database (CMDB), [70, 139, 248, 269](#)
- change management, [156](#)
 - incident management, [156](#)
 - ITIL-driven services, [155](#)
 - provisioning environments, [156](#)
- Configuration management system (CMS), [70, 139–140](#)
- Content management database, [235](#)
- Continual service improvement (CSI), [55–56, 107](#)
- Continuous deployment process, [28](#)
- change adaption, [259–262](#)
 - vs.* continuous delivery, [29](#)
 - release management, [71, 284](#)
- Core service, [43–44](#)
- Cucumber, [34](#)
- ## D
- Database administrator (DBA), [19](#)
- Definition of done (DOD), [181, 184](#)
- Definition of ready (DOR), [181](#)
- Definition of IT service, [42](#)
- Definitive media library (DML), [140–141, 281](#)
- Definitive spares (DS), [141](#)
- Delivery teams, [290](#)
- Demand management process
- alignment, [83](#)
 - DevOps project, [83–84](#)
 - service-based organizations, [82](#)
 - sprint planning, [84](#)
- Design coordination
- availability management, [91](#)
 - point-based design, [86–87](#)
 - service level management, [90](#)
 - set-based design, [87–88](#)

- Developer (DEV), 19
- DevOps
- adoption, 100
 - Agile, 5–6
 - automation, 10–11
 - CALMS feature, 9
 - culture, 3–4, 10
 - definition, 2
 - elements, 13–16
 - lean, 11
 - measurement, 12
 - people
 - CAB, 16
 - development and operations, 16, 17
 - DevOps team, 18–20
 - rollback, 3
 - scope, 6
 - sharing, 13
 - software development, 1
 - technology, 30–34
 - transformation benefits, 7–8
 - waterfall project management, 1, 4–5
- DevOps change management
- process, 256
 - Agile methodology, 248
 - change managers, 249
 - continuous delivery
 - build and test, 258
 - change governance, 259
 - change initiation, 257–258
 - deployment and verification, 258
 - deployment authorization, 258
 - technical teams, 257
 - continuous deployment
 - automated approach, 259
 - build and test, 261
 - change initiation and authorization, 260–261
 - change verification and closure, 261
 - production, 261
 - enterprise change manager, 248
 - organizations, 249
 - project change management
 - (*see* Project change management)
 - rescue, 250
 - risk mitigation strategies
 - auto-deployment and auto-checks, 255–256
 - software quality, 254
 - standard changes (*see* Standard changes)
- DevOps incident management process
- accepting, 194
 - analysis, escalation, and resolution, 193
 - auto deployment, 197–198
 - continuous integration and continuous testing, 197
 - DevOps team, 193–194
 - identification, 193
 - incident manager, 194
 - post-mortem, 198–199
 - prioritization and sprint, 195
 - Scrum team, code changes and check-in, 196–197
 - sequence of activities, 191–192
- DevOps model
- application management function, 125
 - objectives, 125
 - organization's strategy, 128
 - role mapping, 128
 - scrum team, 125
 - team composition, 126
 - team scope, 127

INDEX

DevOps problem management

process, 226–227

auto-deployment, 229

axe sharpening, 224

bugs, 223

continuous integration and testing, 229

major incidents, 224

problem detection, 228

problem manager, 225–226

product backlog, Scrum master, 228–229

product owner, sprint backlog, 229

repetitive incidents, 224

Scrum master, 224

Scrum team, 229

DevOps testing process, 102

Digital age, 41

Distributed version control system (DVCS), 30, 159–160

Dynamic configuration management, 136

E

Early life support, 277–278

Emergency change advisory board (ECAB), 245–246

Emergency changes, 239

Emergency releases, 277

Employee attraction, 9

Enabling service, 43, 45

Enhancing service, 43, 45

Event management, 218, 220

exception, 105

information, 105

warning, 105

F

Facilities management, 117

Financial management process, 82

Flat hierarchy, 121–122

Functions

application management teams, 115–116

facilities management, 117

IT operations management, 116

list of, 47–48

vs. processes, 48–49

service desk, 112–113

technical management teams, 113–114

G

Git, 32

Google Compute Engine, 32

H

HP LoadRunner, 34

I

Incident management, 106, 163

cable TV, Internet, and electricity, 164

categorization, 171–172

closure, 174

diagnosis and investigation, 172–173

downtime reduction, 164

identification

e-mail/chat, 170

event management, 170

telephone, 170

- Web interface, 170
- knowledge
 - (*see* Knowledge management)
- levels of support, 185–187
- logging, 170–171
- major, 174–175
- monitoring tools, 167, 168
- objectives and principles, 165–166
- one team concept, 176
- prioritization, 172
- resolution and recovery, 173–174
- service desk, 165
- service provider, 164
- services, 166
- typical process, 168–169
- users reporting, 167
- vital role, 164–165
- Incident tickets, 171
- Information security management
 - CIA, 97
 - IT innovation, 97
 - Rugged DevOps Manifesto, 98
 - security, 97
- Infrastructure-as-a-service (IaaS), 149
- Infrastructure as code (IaC), 117
- Internet service provider (ISP), 42, 43
- Ishikawa diagram
 - fishbone models, 211–212
 - on-time flight arrival rate, 213–214
 - service and marketing industry, 212
- ITIL phases
 - analysis, 77
 - features, 78–79
- ITIL service lifecycle, 49–50
- ITIL versions, 39–40
- IT operations management, 116
 - facilities management, 117–118
 - operations control, 117
 - IT security (SEC), 20
 - IT service continuity management (ITSCM), 95–96
 - IT service management, 37–38
- J**
- Jenkins, 33
- K, L**
- Kepner-Tregoe method, 215–216
- Key performance indicators (KPIs), 90
- Knowledge management, 103–104
 - configuration management, 187
 - design documents, 188
 - ISO certifications, 188
 - maintaining documents, 189–190
 - minimum documentation, 188
 - service delivery, 188
 - storing and retrieval, 190–191
- Knowledge management database (KMDB), 247
- Known error database (KEDB), 205, 222
- Known errors, 204–205
- M**
- Major incidents, 174–175, 218
- Major releases, 276
- Matrix organization, 114
- Mercurial, 32
- Minimum viable product (MVP), 88, 103
- Minority Report, 219
- Minor releases, 277

INDEX

N

Normal change management process, 238
assess and evaluate, 243
authorization, 247
building and testing, 247
CAB, 243–244
ECAB, 245–246
SCAB, 246
implementation and verification, 247
review and close, 247–248
RFC, 242
workflow, 240–241

O

Office of Government Commerce (OGC), 38
One-click server creation, 149
Operational level agreement (OLA), 90
Operations team, 291
OS-level configurations, 150

P, Q

Passive monitoring, 168
Personal digital assistants (PDAs), 52
Phased deployment, 278–280
Phases of release management
deployment, 281–282
release and deployment
planning, 280
release build and test, 281
review and close, 282
Planning poker, 180–181
Post-implementation review (PIR), 247
Predictability, 124
Proactive problem management, 228
Problem analysis techniques

brainstorming, 206–207
five-why technique
financial mismanagement, 210
flight delays, 208–209
limitations, 210
root-cause, 208
Ishikawa diagram, 211–215
Kepner-Tregoe
method, 215–216
Problem management, 107, 201
categorization, 220
closure, 222
CSI, 202
vs. incidents, 203
investigation and diagnosis, 221
KEDB, 205
known errors, 204–205
logging
analysis/trending, 220
attributes, 220
event management, 220
partners/suppliers, 220
objectives and principles, 202–203
permanent solution, 205
prioritization, 221
problem detection
analysis/trending, 219
event management, 218
major incidents, 218
partners/suppliers, 219
RCA, 204
resolution, 222
root cause, 204
workaround, 205
workflow, 216–217
Problem-solving and decision-making
(PSDM) technique, 215

- Processes
- Agile, 21
 - automation *vs.* continuous testing, 26–27
 - continuous delivery, 24–26
 - continuous deployment, 28–29
 - continuous integration, 21–24
 - definition, 46
 - and functions, 47–49
- Process manager, 58–59
- Process owner, 58
- Process practitioner, 59
- Product backlog, 179
- Product owners (POs), 19, 72, 123, 253, 294–295
- Product *vs.* services
- DevOps implementations, 63, 67
 - IT magnet, 64–65
 - solution provider, 65–66
 - XaaS model, 66
- Project change management
- ARTs, 251
 - change control, 250–251
 - cost, 251, 253
 - quality, 254
 - resource allocation, 252
 - schedule, 253
 - scope, 251
- Proprietary knowledge, 41
- Puppet, 33
- R**
- Reactive problem management, 228
- Release and deployment management, 101, 273
- Release management, 71
- adaption
 - ART, 284
- continuous delivery, 285
- continuous deployment, 284
- automation, 288–289
- blue-green deployment, 286–287
- vs.* change management, 271–273, 285
- deployment options
- Big Bang, 278–279
 - phased approach, 278–280
- early life support, 277–278
- emergency releases, 277
- iterative and sequential phases, 282–283
- major releases, 276
- minor releases, 277
- vs.* release and deployment management, 273
- release package, 275–276
- release unit, 274–275
- scope of, 287
- teams
- delivery, 290
 - development and operations, 291–294
 - Jira, 290
 - operations, 291
 - productivity, 289
 - product owners (POs), 294–295
 - release manager, 291
 - separate release, 290
- Request for change (RFC), 242
- Request fulfillment process, 106
- Responsible, Accountable, Consulted, and Informed (RACI) matrix, 59–61
- Role mapping
- DevOps teams scope, 132–133
 - shared teams scope, 131–132
 - strategy and compliance, 129–130
 - umbrella teams scope, 130–131
- Root cause, 204
- Root-cause analysis (RCA), 204, 221

S

Scaled Agile Framework (SAFe), 86
 Scrum master (SM), 19, 122, 195, 224,
 228–229, 294
 Selenium, 34
 Separate release management team, 290
 Sequential *vs.* concurrent, 68
 Service asset and configuration
 management (SACM), 89, 101
 accounting, 145
 control process, 144–145
 elements, 143
 identification, 144
 ITIL, 135
 models, 142
 planning, 143
 principles, 136
 reporting, 145
 scope, 138
 verification and audit, 146
 Service capacity management, 94
 Service catalog management, 89
 Service design, 52–53
 capacity management, 92
 design coordination, 85
 information security management, 96
 ITSCM, 95
 service catalog management, 89
 supplier management, 99
 Service desk
 functional escalation, 113
 integral component, ITIL, 112
 Service integration and management
 (SIAM), 99
 Service level agreements (SLAs), 43, 74, 90
 Service level management, 90

Service level requirements (SLRs), 90
 Service management framework, 69, 77
 Service manager (SMG), 20
 ServiceNow, 242
 Service operations, 54–55
 access management, 107
 event management, 104–105
 incident management (*see* Incident
 management)
 problem management
 (*see* Problem management)
 Service owner, 57
 Service portfolio management, 82
 Service strategy, 50–52
 business relationship management, 84
 demand management, 82–84
 IT services, 79–81
 financial management, 82
 service portfolio management, 82
 Service transition, 53–54
 change evaluation, 102–103
 change management (*see* Change
 management)
 knowledge management, 103
 release and deployment
 management, 101
 SACM (*see* Service asset and
 configuration management
 (SACM))
 transition planning and support, 100
 validation and testing, 101–102
 Set-based design (SBD), 86
 Seven-step improvement process,
 108–109
 Silo culture, 69–70
 Single point of failure (SPOF), 30
 Software delivery lifecycle (SDLC), 1–24

- Source code management (SCM), 70, 157
- Source code repository (SCR)
- DevOps objectives, 158
 - storage, 157
 - tools
 - CVCS, 159, 161
 - DVCS, 159–161
 - version control systems, 157
- Sprint backlog, 179, 195
- Sprint planning
- capacity and velocity, 179
 - complexity, 180
 - DevOps team
 - contingency, 184
 - input funnel, 182
 - SLA, 182–183
 - story points, 183
 - user stories, 183
 - DOR and DOD, 181–182
 - planning poker, 180–181
 - product backlog, 179
- Sprint retrospective, 124
- Standard change advisory board (SCAB), 246, 267–268
- Standard changes, 239–240
- advantage, 262
 - Agile and DevOps, 264
 - assessment, 263
 - candidates, 267
 - developing change models, 268
 - enterprise change manager, 263
 - fatal damages, 262
 - identifying and managing, 264–265
 - implementation, 268–269
 - monitoring and auditing, 269–270
 - nominating candidates, 266
- qualification, 266
 - SCAB, 267
 - service management process, 262
- State of DevOps Report, 8–9
- Story point estimation, 180
- Strategy management
- big-bang approach, 81
 - IaC model, 81
 - service development in the Agile, 81
- Supplier management, 99
- System administrator (SYS), 20
- ## T
- Team structure
- agile model, 121
 - decision-makers, 118
 - DevOps model, 125
 - traditional model, 119
- Technical management teams, 113–114
- Technical service catalog, 89
- Technology
- CVCS, 30
 - deployment and environment
 - provisioning, 33
 - DVCS, 30
 - hosting services, 32
 - orchestrators, 33
 - periodic table, 31
 - source code repositories, 32
 - testing, 34
 - tools, 30–34
- Testers (TEST), 19
- Toyota Production Systems (TPS), 11
- Traditional model
- matrix organization, 119
 - mobilizing, project team, 120

INDEX

U, V

Unauthorized changes, [235–236](#)
Union of mind-sets, [72–73](#)
UrbanCode Deploy, [33](#)
User acceptance testing
(UAT), [281](#)

Users report incidents, [167](#)
User stories, [176–177](#)

W, X, Y, Z

Waterfall model, [5, 73, 282](#)
Workaround, [205](#)