



Network Automation Made Easy



Network Automation Made Easy

Ivo Pinto [CCIE No. 57162]

Cisco Press

Network Automation Made Easy

Ivo Pinto

Copyright © 2022 Cisco Systems, Inc.

Cisco Press logo is a trademark of Cisco Systems, Inc.

Published by:

Cisco Press

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions.

No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ScoutAutomatedPrintCode

Library of Congress Control Number: 2021915881

ISBN-13: 978-0-13-750673-6

ISBN-10: 0-13-750673-2

Warning and Disclaimer

This book is designed to provide information about network automation, it covers the current landscape, practical applications, tools, and techniques. Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied.

The information is provided on an “as is” basis. The authors, Cisco Press, and Cisco Systems, Inc. shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the

information contained in this book or from the use of the discs or programs that may accompany it.

The opinions expressed in this book belong to the author and are not necessarily those of Cisco Systems, Inc.

Feedback Information

At Cisco Press, our goal is to create in-depth technical books of the highest quality and value. Each book is crafted with care and precision, undergoing rigorous development that involves the unique expertise of members from the professional technical community.

Readers' feedback is a natural continuation of this process. If you have any comments regarding how we could improve the quality of this book or otherwise alter it to better suit your needs, you can contact us through email at feedback@ciscopress.com. Please make sure to include the book title and ISBN in your message.

We greatly appreciate your assistance.

Editor-in-Chief

Mark Taub

Alliances Manager, Cisco Press

Arezou Gol

Director, ITP Product Management

Brett Bartow

Managing Editor

Sandra Schroeder

Development Editor

Ellie C. Bru

Project Editor

Mandie Frank

Copy Editor

Kitty Wilson

Technical Editors

Asier Arlegui Lacunza; Celia Ortega Gomez

Editorial Assistant

Cindy Teeters

Designer

Chuti Prasertsith

Composition

codeMantra

Indexer

Proofreader

Trademark Acknowledgments

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Cisco Press or Cisco Systems, Inc. cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

About the Author

Ivo Pinto, CCIE No. 57162 (R&S, Security, and Data Center), CISSP, is a Solutions Architect with many years of experience in the fields of multicloud, automation, and enterprise and data center networking. Ivo has worked at Cisco in different roles and different geographies, and he has led the architecture creation and deployment of many automated global-scale solutions for Fortune 50 companies that are in production today. In his latest role, he is responsible for multicloud innovation at the Customer Experience CTO office. Ivo is the founder of IT OnTrack (www.itontrack.com), a provider of services to young professionals who are looking to further their careers in IT. Ivo has authored Cisco white papers and multiple Cisco exam questions.

Follow Ivo on LinkedIn @ivopinto01.

About the Technical Reviewers

Asier Arlegui Lacunza, CCIE No. 5921, has been with Cisco since 1998 and currently works as a Principal Architect in the Cisco Customer Experience organization. In the past 20+ years of his career at Cisco, he has worked as a technical architect on a wide range of enterprise (data center, campus, and enterprise WAN) and service provider (access and core networking) technology projects, with a focus on network automation. He holds a master's degree in telecommunications engineering from Public University of Navarra, Spain.

Celia Ortega Gomez joined Cisco in the Network Consulting Engineer Associate Program in 2017. She has been working as a consulting engineer, focusing on network automation, routing and switching, and visibility technologies in the Cisco Customer Experience organization for EMEAR customers. She holds several industry certifications and is now working as the Manager for South PMO Office and High Touch services.

Dedications

I would like to dedicate this book to my family, who has supported me on every step of this long journey. To my loving mother, thank you for believing that one day I would write a book. To my wonderful wife, thank you for the tireless encouragement throughout the process. And last but not least, to my two cats, Oscar and Tofu, who kept me company during the countless all-nighters I pulled to finish the book.

Acknowledgments

Special thanks to the technical reviewers, Asier Arlegui and Celia Ortega Gomez, who contributed to a substantial increase in quality not only with their corrections but also suggestions to the content of this book. Also, special thanks to Pedro Marques, Pedro Trigueira, and Miguel Santos for their guidance and advice throughout the process. I would like to thank my former and my current leadership at Cisco, Stefaan de Haan, Mark Perry, Carlos Pignataro, Adele Trombeta, and Steve Pickavance, for supporting me during the lengthy process of writing a book.

This book wouldn't have been possible without the support of many people on the Cisco Press team. Brett Bartow, Director, Pearson IT professional Group, was instrumental in sponsoring the book and driving it to execution. Eleanor Bru, Development Editor, has done an amazing job in the technical review cycle, and it has been an absolute pleasure working with you. Also, many thanks to the numerous Cisco Press unknown soldiers working behind the scenes to make this book happen.

Contents at a Glance

Introduction

Chapter 1. Types of Network Automation

Chapter 2. Data for Network Automation

Chapter 3. Using Data from Your Network

Chapter 4. Ansible Basics

Chapter 5. Using Ansible for Network Automation

Chapter 6. Network DevOps

Chapter 7. Automation Strategies

Appendix A. Answers to Review Questions

Contents

Introduction

- Goals and Methods
- Who Should Read This Book?
- How This Book Is Organized
- Book Structure

Chapter 1. Types of Network Automation

- Data-Driven Automation
- Task-Based Automation
- End-to-End Automation
- Tools
- Summary
- Review Questions

Chapter 2. Data for Network Automation

- The Importance of Data
- Data Formats and Models
- Methods for Gathering Data
- Summary
- End Notes
- Review Questions

Chapter 3. Using Data from Your Network

- Data Preparation
- Data Visualization
- Data Insights

[Case Studies](#)
[Summary](#)
[Review Questions](#)

Chapter 4. Ansible Basics

[Ansible Characteristics](#)
[Installing Ansible](#)
[Variables](#)
[Playbooks](#)
[Conditionals](#)
[Loops](#)
[Handlers](#)
[Executing a Playbook](#)
[Roles](#)
[Summary](#)
[Review Questions](#)

Chapter 5. Using Ansible for Network Automation

[Interacting with Files](#)
[Interacting with Devices](#)
[Interacting with APIs](#)
[Case Studies](#)
[Summary](#)
[Review Questions](#)

Chapter 6. Network DevOps

[What NetDevOps Is](#)
[NetDevOps Tools](#)
[How to Build Your Own NetDevOps Environment](#)
[Case Studies](#)

[Summary](#)

[Review Questions](#)

Chapter 7. Automation Strategies

[What an Automation Strategy Is](#)

[Why You Need an Automation Strategy](#)

[How to Build Your Own Automation Strategy](#)

[Summary](#)

[Review Questions](#)

Appendix A. Answers to Review Questions

Icons Used in This Book



placeholder

Command Syntax Conventions

The conventions used to present command syntax in this book are the same conventions used in Cisco's Command Reference. The Command Reference describes these conventions as follows:

- **Boldface** indicates commands and keywords that are entered literally as shown. In actual configuration examples and output (not general command syntax), boldface indicates commands that are manually input by the user (such as a **show** command).
- *Italics* indicate arguments for which you supply actual values.
- Vertical bars (|) separate alternative, mutually exclusive elements.
- Square brackets [] indicate optional elements.
- Braces { } indicate a required choice.
- Braces within brackets [{ }] indicate a required choice within an optional element.

Introduction

From the moment networks appeared, they steadily grew bigger and became increasingly complex. Engineers have sought to automate any possible activity since then. Network automation is the process of automating the configuration, management, testing, deployment, and operation of physical and virtual devices in a network. By eliminating tedious and manual processes, it is possible to reduce operational expenses, reduce human error, and achieve better time to market.

Network automation is not a new topic, but in recent years, it has exponentially increased in importance due to external factors such as the need for agility in responding to growth. Although it's still in its infancy, network automation is already a critical pillar on an organization's strategy. Companies are redesigning and rethinking their network strategies, and some are even being pushed toward automation without clear processes or goals.

This book approaches the topic from the point of view of an IT professional who is well versed in networking and related topics—including cloud, compute, and other components in today's networks—and is trying to take both physical and virtual infrastructure to a semi- or fully automated state. The book explains the fundamentals of network automation, including a variety of tools and their use cases and data and how to extract value from it. This book also takes a deep dive into a specific tool, Ansible, showing how to use it to solve a number of common use cases in networking.

A key aspect of the book is its practical approach to the topic. It is filled with code snippets you can reuse for your own use cases, as well as real case studies that show practical applications of all this knowledge.

Although this is a Cisco Press book, it takes a vendor-neutral approach to automation tools and techniques. It will give you the knowledge you need to make informed decisions when automating your own use cases.

Last but not least, the book explains how to produce a network automation strategy, which is a key piece that is notoriously missing in many enterprises

today. It helps a reader focus automation efforts on a fruitful network automation journey rather than a journey without a clear purpose.

Goals and Methods

IT professionals are now more than ever before challenged by their businesses to meet a level of network agility and elasticity that only automation can solve. Furthermore, as networks have expanded to the cloud, they have become increasingly dynamic and complex. To address this, vendors introduce new tools every other day, many of them overlapping in the problems they address. This makes it very challenging for the IT professionals to choose the correct tool for their use case.

The goal of this book is to help you understand what can and should be automated in your network environment, what benefits automation would bring, and how you can achieve it. Furthermore, it compares and contrasts some of the available tools to help you understand where to use each of them and why. This knowledge will allow you to make informed network automation decisions in your company.

Who Should Read This Book?

Network automation typically touches several network components, such as routers, switches, firewalls, virtual machines, and cloud infrastructure. In general, IT professionals are divided in their areas of expertise. Individuals are spread into focus areas such as the following, which in some cases overlap:

- Servers and virtualization
- Storage
- Switching and routing
- Security
- Software applications

- Cloud

As the focus of this book is network automation, the audience is the sum of all system administrators, storage administrators, networking engineers, software virtualization engineers, and network management engineers.

Because this book also approaches real case studies and network automation strategy, IT managers will also benefit from reading it as it will help them understand how automation can greatly improve their ecosystem and how to plan for the journey.

How This Book Is Organized

This book is set up to help you understand and replicate the use cases on your own. It is recommended that you read through the chapters in order to get the full benefit of the book.

Networking, storage, compute, virtualization, and cloud are complex topics and are getting more complex every day. System administrators, networking engineers, cloud engineers, and virtualization engineers are asked to master their field and also to automate the solutions on their field. This book helps you understand the fundamentals of network automation and apply it to your job needs, using state-of-the-art tools and techniques. This book offers a number of advantages:

- An easy reading style with no marketing
- Comprehensive coverage of the topic, from fundamentals to advanced techniques
- Real case studies, instead of hypothetical situations, of projects the author led.
- Coverage of the latest network automation trends, such as NetDevOps
- Reusable code snippets
- Explanations of tools and their applications with neutrality

This book is beneficial to IT professionals trying to understand how to implement network automation solutions as well as to IT management trying to understand the benefits of network automation and where to apply it.

Book Structure

The book is organized into seven chapters:

- **Chapter 1, “Types of Network Automation”:** This chapter describes the different network automation types and the use cases they address. This chapter also compares and contrasts some of the common automation tools used by modern enterprises, including their advantages and shortcomings, in a vendor-neutral way.
- **Chapter 2, “Data for Network Automation”:** This chapter provides fundamental background on data and the role it plays in network automation. The chapter describes commonly seen data formats, such as JSON, XML, and YAML, in detail. Finally, it consolidates the topic by describing methods and techniques to gather data from your network.
- **Chapter 3, “Using Data from Your Network”:** This chapter provides an overview of what you can do with the data that you gather from your network. The chapter starts by defining common data preparation techniques, such as parsing and aggregation, followed by data visualization techniques. Finally, it describes possible insights you can derive from your gathered data. The chapter also describes three case studies about enterprises that have created notable value from network data.
- **Chapter 4, “Ansible Basics”:** This chapter examines Ansible, which is the most commonly used automation tool in enterprises. It describes all the fundamental building blocks of Ansible, including the tool’s installation as well as its architecture and components, such as plays, tasks, modules, variables, conditionals, loops, and roles.
- **Chapter 5, “Using Ansible for Network Automation”:** This chapter examines Ansible as a network automation tool. It illustrates how to achieve common networking tasks with Ansible playbooks across a variety of components, such as files, networking devices, virtual machines, cloud constructs, and APIs. Furthermore, the chapter describes three use cases where Ansible was used to automate global-scale network architectures.

- **Chapter 6, “Network DevOps”:** This chapter introduces NetDevOps and how it is changing the automation paradigm in networking. It covers the fundamentals of what NetDevOps is, why you would use it, and where you would use it. The chapter also guides you through the step-by-step process of creating your own NetDevOps pipeline. The chapter finishes with three case studies that give you a feel for the benefits NetDevOps has brought to some large enterprises.
- **Chapter 7, “Automation Strategies”:** This chapter defines network automation strategy and delves into its functionality. It provides a detailed description of what an automation strategy is and its components. This chapter also includes a methodology to build an automation strategy from scratch, along with tips and lessons learned from applying automation in large enterprises. It finishes with an overview of how to plan the execution of a newly created strategy.

Chapter 1. Types of Network Automation

Automation has taken the world of networking by storm in the past couple years. The speed and agility automation provides to companies is unprecedented compared to the “old way” of doing things manually. In addition to speed, automation can bring many other benefits to networking, such as consistency in actions and reduction of human error.

Automation can be applied to many different processes and technologies in different ways in the world of networking. In this book, when we refer to *network automation*, we are referring to the process of automatic management, configuration, testing, deployment, and operation in any kind of network environment, either on premises or in the cloud. This chapter approaches automation by subdividing it into the following types and providing use cases and tools for each of them:

- Data-driven automation
- Task-based automation
- End-to-end automation

This division is not an industry standard but is a reflection of the most commonly seen outcomes from network automation.

Note

This is not a deeply technical chapter but rather an introductory one that describes automation architectures and common use cases. It lays the foundations for the upcoming chapters, which go more in detail on some of these topics.

[Figure 1-1](#) provides an overview of the use cases described for each of these

three types of automation in this chapter.

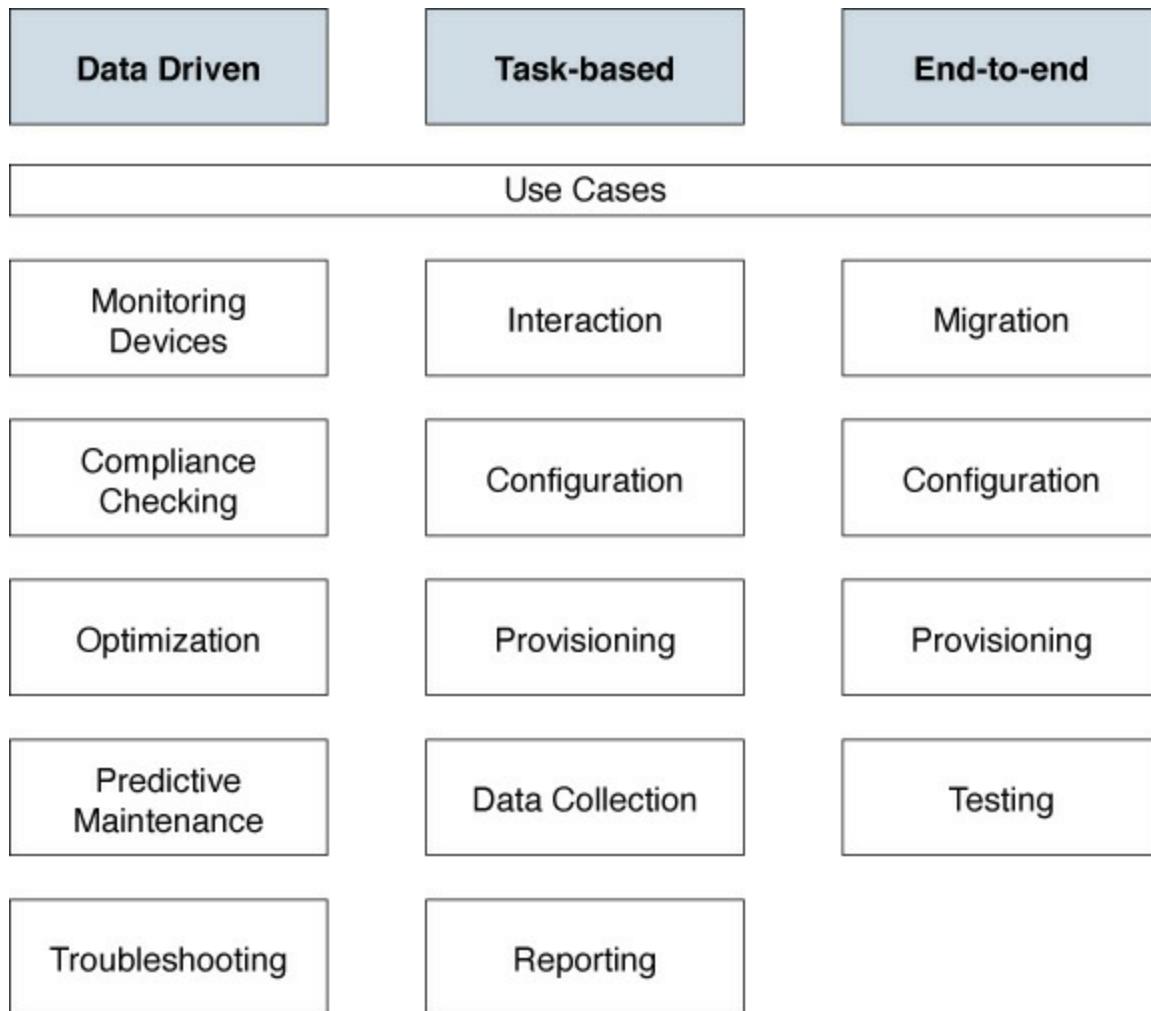


Figure 1-1 Automation Use Cases Overview

Data-Driven Automation

Everyone is talking about data, automation folks included. Data is undoubtedly among the most important components in a system. But what actually is data? *Data* is facts or statistics collected about something.

Data can be used to understand the state of a system, used to make decisions based on reasoning and/or calculations, or stored and used later for comparisons, among numerous other possibilities.

The possible insights derived by analyzing data make data an invaluable asset also for networking automation solutions.

In [Chapter 2](#), “[Data for Network Automation](#),” you will see how to capture data for networking purposes, and in [Chapter 3](#), “[Using the Data of Your Network](#),” you will learn how to prepare and use the captured data.

What Data-Driven Automation Is

Data-driven automation can be involves automating actions based on data indicators. As a simple example, you could use data-driven automation to have fire sprinklers self-activate when the heat levels are abnormally high. Data-driven automation is the prevailing type of automation in the networking world. Firewalls can be configured to block traffic (action) from noncompliant endpoints (compliance data), and monitoring systems can trigger alarms (action) based on unexpected network conditions such as high CPU usage on networking equipment (metric data).

Data-driven automation can take many shapes and forms, but it always *starts from data*. Data can be expressed in many formats, and we cover common industry formats in [Chapter 2](#).

There are push and pull models in data-driven automation architectures. In push models, devices send information to a central place. The sensors in [Figure 1-2](#), for example, send measurements to the management station, which receives them and can take action based on those readings. Examples of networking protocols that use a push model include Syslog, NetFlow, and SNMP traps.

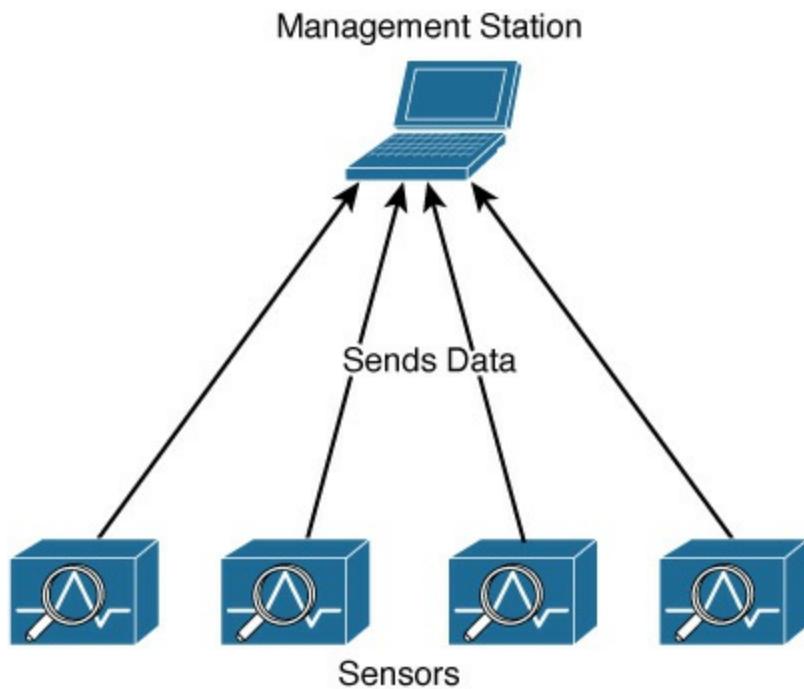


Figure 1-2 Push Model Sensor Architecture

Pull models (see [Figure 1-3](#)) are the opposite of push models. In a pull model, the management station polls the sensors for readings and can take action based on the results. A networking protocol that utilizes a pull model is SNMP polling.

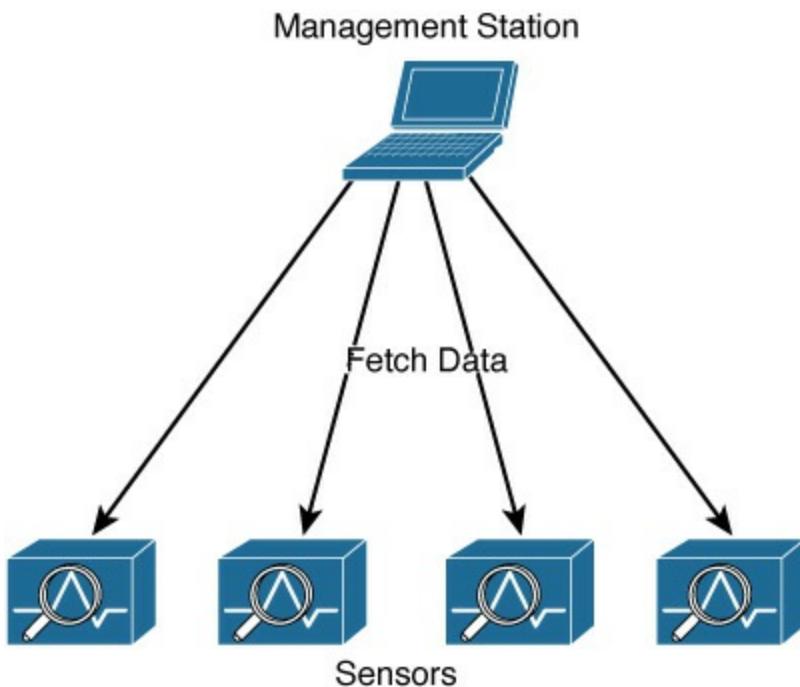


Figure 1-3 Pull Model Sensor Architecture

To better understand the difference between these two models, consider a real-life example involving daily news. With a pull model, you would go to a news website to read news articles. In contrast, with a push model, you would subscribe to a news feed, and the publisher would push notifications to your phone as news stories are published.

Push models are popular today because they allow for faster actions as you don't have to pull data at a specific time; [Chapter 2](#) covers model-driven telemetry, which is an example of a push model. Modern push protocols allow you to subscribe to specific information, which will be sent to you as soon as it is available or at configurable fixed time intervals. However, in some use cases, the pull model is preferred, such as when the computing power available is limited or there are network bandwidth constraints. In these cases, you would pull only the data you think you need at the time you need it. The problem with a pull model is that it may not always give you the information you need. Consider a scenario in which you are polling a temperature measurement from a network device; failing to poll it regularly or after an event that would increase it could have disastrous consequences.

You have probably been using data-driven automation for years. The

following section guide you through use cases for adding automation to a network.

Data-Driven Automation Use Cases

Even if they don't use the term *automation*, companies use data-driven automation all the time. There are opportunities for automation in many IT processes. The following sections describe some use cases for data-driven automation. Later in the book you will see case studies of some of these use cases transformed and applied to global-scale production systems.

Monitoring Devices

One of the quickest and easiest ways of getting into automation is by monitoring device configurations or metrics.

Monitoring device metrics is a well-established field, and you can use a number of tools that can collect network statistics and create dashboards and reports for you. We cover some of these tools later in this chapter; in addition, you can easily create your own custom solutions. One of the reasons to create your own tools is that off-the-shelf tools often give you too much information—and not necessarily the information you need. You might need to build your own tool, for example, if you need to monitor hardware counters, because industry solutions are not typically focused on specific detailed implementations. By collecting, parsing, and displaying information that is relevant for your particular case, you can more easily derive actionable insights.

Monitoring device configurations is just as important as device metrics as it prevents *configuration drift*—a situation in which production devices' configurations change so that they differ from each other or from those of backup devices. With the passing of time, point-in-time changes are made to deal with fixes or tests. Some of the changes are not reflected everywhere, and thus drift occurs. Configuration drift can become a problem in the long run, and it can be a management nightmare.

The following are some of the protocols commonly used in network device monitoring:

- Simple Network Management Protocol (SNMP) (metrics)
- Secure Shell Protocol (SSH) (configuration)
- NetFlow (metrics)
- Syslog (metrics)

Each of these protocols serves a different purpose. For example, you might use SNMP to collect device statistics (such as CPU usage or memory usage), and you might use SSH to collect device configurations. [Chapter 3](#) looks more closely at how to collect data and which methods to use.

Automating network monitoring tasks does not simply add speed and agility but allows networking teams to get updated views of their infrastructure with minimal effort. In many cases, automating network monitoring reduces the administrative burden.

To put this into perspective, let's look at an example of a network operations team that needs to verify that the time and time zone settings in all its network devices are correct so that logs are correlatable. In order to achieve this, the network operator will have to follow these steps:

Step 1. Connect to each network device using SSH or Telnet.

Step 2. Verify time and time zone configurations and status.

- a. If these settings are correct, proceed to the next device.
- b. If these settings are incorrect, apply needed configuration modifications.

The operator would need to follow this process for every single device in the network. Then, it would be a good idea to revisit all devices to ensure that the changes were applied correctly. Opening a connection and running a command on 1000 devices would take a lot of time—over 40 hours if operators spend 2 or 3 minutes per device.

This process could be automated so that a system, rather than an operator, iterates over the devices, verifies the time and time zone settings, and applies configuration modifications when required. What would take 40 hours as a manual process would take less than 2 hours for automation, depending on

the tool used. In addition, using automation would reduce the possibility of human error in this process. Furthermore, the automation tool could later constantly monitor time data and alert you when time is not in sync. From this example, you can clearly see some of the benefits of automation.

The process used in the time verification example could also be used with other types of verification, such as for routing protocols or interface configurations. You will see plenty of examples throughout this book of applying data-driven automation using tools such as Python and Ansible.

Compliance Checking

Most network engineers must deal with compliance requirements. Systems must be built for regulatory compliance (for example, PCI DSS, HIPAA) as well as for company policy. In many cases, compliance checking has become so cumbersome that some companies have dedicated roles for it, with individuals going around monitoring and testing systems and devices for compliance.

Today, we can offload some of the burden of compliance checking to automated systems. These systems systematically gather information from target systems and devices and compare it to the desired state. Based on the results, a range of actions can be taken, from a simple alert to more complex remediation measures.

[Figure 1-4](#) illustrates a simple data-driven automated flow:

- Step 1.** The management system polls device configuration by using a protocol such as SSH.
- Step 2.** The management system compares the device configuration against predefined template rules.
- Step 3.** The management system pushes a configuration change if the compliance checks failed.

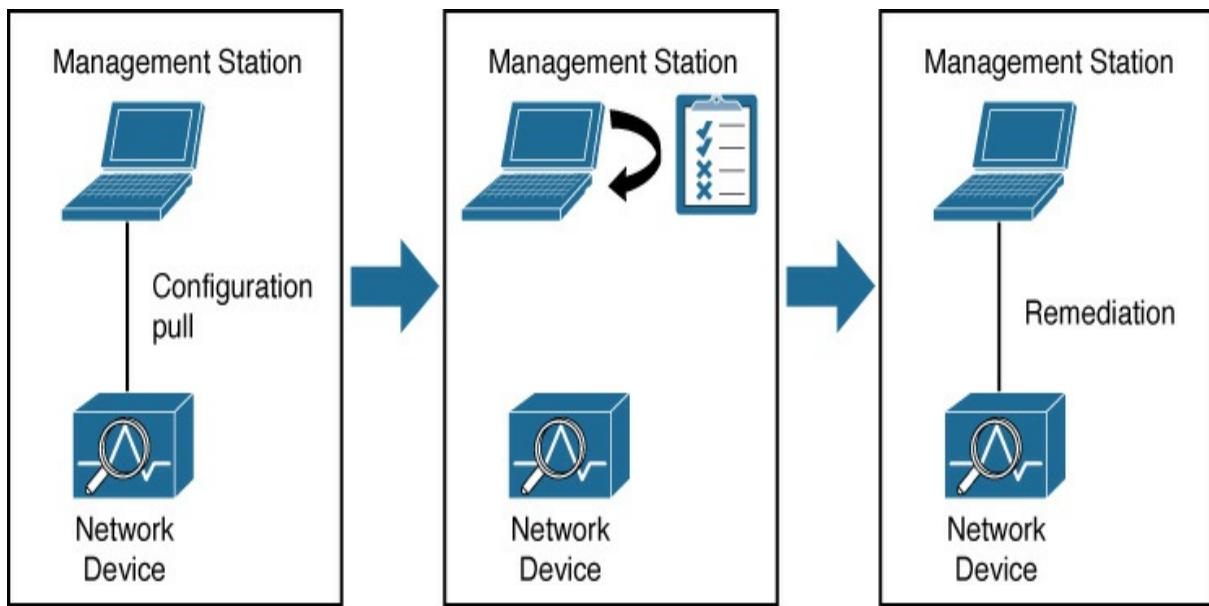


Figure 1-4 *Compliance Workflow*

Automated compliance tools can typically generate reports that keep you aware of the network's compliance status. In addition, integrations with other network solutions can help you secure your network. For example, through integration with an endpoint management system, you can quarantine noncompliant endpoints, reducing the network's exposure in the event of a cyberattack.

How are you doing your compliance checks today? If you are already using a tool for compliance checks, you are already on the automation train! If you are doing these checks manually, you might consider implementing one of the tools presented later in this chapter to help you with the task.

Optimization

Organizations are eager to optimize their networks—to make them better and more efficient. Optimization is an activity that we often do manually by understanding what is currently happening and trying to improve it. Possible actions are changing routing metrics or scaling elastic components up and down. Some automation tools can do such optimization automatically. By retrieving data from the network constantly, they can perceive what is not running as efficiently as possible and then take action. A good example of automated network optimization is software-defined wide area networking

(SD-WAN). SD-WAN software constantly monitors WAN performance and adjusts the paths used to access resources across the WAN. It can monitor metrics such as latency to specific resources and link bandwidth availability. It is a critical technology in a time when many of the resources accessed are outside the enterprise domain (such as with SaaS or simply on the Internet).

Imagine that you have two WAN links, and your company uses VoIP technologies for critical operations that, when disrupted, can cost the business millions of dollars. Say that you configure a policy where you indicate that packet loss must be less than 3% and latency cannot exceed 50ms. An SD-WAN controller will monitor the WAN links for predefined metrics and compare them to your policy. When the controller detects that your main WAN link is experiencing poor performance for voice traffic, it shifts this type of traffic to your second link. This type of preventive action would be considered an optimization.

As another example of optimization, say that you have some tools that gather the configurations from your network devices and systems. They compare them to configuration best practices and return to you possible enhancements or even configuration issues. Optimization tools today can even go a step further: They can also collect device metrics (for example, link utilization, client density) and compare them to baselines of similar topologies. These tools can, for example, collect data from several customers, anonymize that data, and use it as a baseline. Then, when the tool compares device metrics against the baseline, it might be able to produce insights that would otherwise be very difficult to spot.

Predictive Maintenance

Isn't it a bummer when a device stops working due to either hardware or software issues? Wouldn't it be nice to know when something is going to fail ahead of time so you could prepare for it?

With predictive maintenance, you can gather data from your network and model systems that attempt to predict when something is going to break. This involves artificial intelligence, which is a subset of automation. Predictive maintenance is not magic; it's just pattern analysis of metrics and logs to identify failure events.

Take, as an example, router fans. By collecting rotation speed over time (see

[Table 1-1](#)), you might find that before failure, router fans rotate considerably more slowly than they do when they’re brand new. Armed with this information, you could model a system to detect such behavior.

Table 1-1 Router Fan Rotations per Minute (RPM)

Fan	T0	T1	T{x}	T{end - 1}	T{end}
Fan 1	1000	980	X	800	0
Fan 2	1000	990	X	799	0
Fan 3	1000	987	X	830	0
Fan 4	1000	991	X	797	0
Fan 5	1000	999	X	803	0
Fan 6	1000	990	X	819	0
Fan 7	1000	987	X	830	0
Fan 8	1000	100	X	900	0
Fan 9	1000	983	X	820	0

Predictive models typically use more than one parameter to make predictions. In machine learning terms, the parameters are called *features*. You may use many features combined to train predictive models (for example, fan RPM, system temperature, device Syslog data). Such a model takes care of prioritizing the relevant features in order to make correct predictions.

Bear in mind that modeling such prediction systems requires a fair amount of data. The data needed depends on the complexity of the use case and the algorithms used. Some models require millions of data points in order to show accurate predictions.

Predictive maintenance is still in its infancy. However, it has been showing improved results lately, thanks to increased computing power at attractive prices. You will see more on machine learning techniques to achieve this sort of use case in [Chapter 3](#).

Another prediction example is load prediction, which can be seen as a type of maintenance. Some tools collect metrics from network devices and interfaces and then try to predict whether a link will be saturated in the future.

Say that you have a branch site with a 1Gbps link to the Internet. Depending

on the time of the day, the link may be more or less saturated with user traffic. Over time, users have been using more external resources because the company has adopted SaaS. You could use a tool to constantly monitor this link's bandwidth and get warnings when you might run into saturation on the link. Such warnings would provide valuable insight, as you would be able to act on this information before you run into the actual problem of not having enough bandwidth for users.

Troubleshooting

Network engineers spend a good portion of their time troubleshooting. Common steps include the following:

Step 1. Connect to a device.

Step 2. Run **show** commands.

Step 3. Analyze the output of these commands.

Step 4. Reconfigure features on the device.

Step 5. Test for success.

Step 6. Repeat steps 1–5 for each of the other devices you need to troubleshoot.

Take a moment to consider how much knowledge you need in order to correlate logs to a root cause or to know what protocol is causing loss of connectivity for a specific user. Now consider that you might need to do this in the middle of the night, after a long workday.

Automation can lighten the load in troubleshooting. By automating some of the aforementioned steps, you can save time and solve problems more easily. Some tools (such as DNA Center, covered later in this chapter) already have troubleshooting automation embedded. The way they work is by deriving insights from collected data and communicating to the operator only the end result. Such a tool can handle the steps 1, 2, and 3; then you, as the operator, could act on the analysis. Steps 4 and 5 could also be automated, but most troubleshooting automation tools are not in a closed-loop automated state, which is a state where the tool collects, analyzes, and acts on the analysis without any input from a human. Rather, with most troubleshooting

automation tools, humans still press that final button to apply the changes, even if it just to tell the tool to apply the changes; sometimes the only human interaction is a simple confirmation.

Visualize the following troubleshooting scenario:

- Reported issues:
 - Applications and users are reporting loss of connectivity.
- Symptoms:
 - Several switches are experiencing high CPU usage.
 - Packet drop counters are increasing.
- Troubleshooting steps:
 - Collect log messages from affected devices by using **show** commands.
 - Verify log messages from affected devices.
 - Trace back activity logs to the time when the problem was first reported.
 - Correlate logs by using other troubleshooting techniques, such as **traceroute**.

After the users report problems, you could follow the troubleshooting steps yourself, but you would need to narrow down the issue first. You could narrow it down by collecting metrics from a network management system or from individual devices. After this initial step, you could troubleshoot the specific devices. While you do all this, users are still experiencing issues.

An automated system is more agile than a human at troubleshooting, and it can make your life easier. An automated system consistently collects logs and metrics from the network, and it can detect a problem when it sees values that do not match the expected values. The system can try to match the symptoms it sees to its knowledge base to identify possible signature problems. When it finds a match, it reports discoveries to the network operator or even mitigates them itself. Most tools report findings rather than resolve them due to the critically some of networks.

In this specific scenario, the underlying issue could be that a Layer 2 loop is

being by a misplaced cable connection. Receiving a notification of a possible Layer 2 loop is much more helpful and actionable than hearing that “some users are reporting loss of connectivity.” This example illustrates the real power of automation: enabling and empowering people to do a better job.

Note

Keep in mind that with troubleshooting automation, it is crucial to have a good knowledge base. Without such a knowledge base, deriving insights from collected data will not be effective.

Task-Based Automation

Tasks are actions. In networking, you have to perform a whole bunch of actions across many devices several times. These actions can be simple or complex, entailing several steps. The need to perform actions can be business derived (for example, implementing a new feature) or technology derived (such as a version upgrade). The bigger the change, the higher the operational burden. Task-based automation attempts to reduce this burden and improve the speed and reliability of tasks.

We discuss how to implement task-based automation using Ansible in [Chapter 4, “Ansible Basics,”](#) and [Chapter 5, “Using Ansible for Network Automation.”](#)

What Task-Based Automation Is

Task-based automation involves automating tasks you would otherwise have to do manually. This category of automation does not need a data input or an automatic trigger to happen; most of the time, it is triggered by an operator.

Network operators often need to create new VLANs for new services. To do this, you have to connect to every device that requires the new VLAN in order to configure it. With task-based automation, you could have a system do this VLAN configuration task for you. Depending on the automation system chosen, you would need inputs such as which devices to configure,

what to configure, and when to configure it.

Task-based type of automation is where most organizations start developing in-house solutions. These are the typical steps to start on your own automation journey:

Step 1. Survey what tasks are commonly executed.

Step 2. Assess the feasibility of automating them and the expected return value.

Step 3. Choose an appropriate tool.

Step 4. Develop use cases on the chosen tool.

As an exercise, think of a task that you commonly do or that your team commonly does. How would you automate it? What tool would you use? You can refer to the “Tools” section at the end of this chapter to help make that decision.

Task-Based Automation Use Cases

This section guides you through some common use cases for task-based automation. Don’t be scared if it seems complex at first. Most complex tasks can be broken down into smaller and less complex subtasks.

Interaction

Interacting with a component is the most generic use case. You can automate almost any interaction you do manually, including these:

- Opening a website
- Interacting with an application
- Inserting data in a database
- Create a spreadsheet in Excel

If you currently do something repeatedly, chances are it is worth thinking about automating it.

As an example of an interaction, say that you are constantly given Excel sheets with network parameters (for example, hostname, IP address, location)

from newly deployed devices in branches. After receiving the sheets by email, you manually insert the parameters in your network management system database via the web GUI available at a URL. This type of activity is time-consuming and does not benefit from intellectual capital. You could automate this flow of interactions and liberate yourself from the administrative burden.

Data Collection

Collecting data is a day-to-day activity for many people. There are many data types and formats, and we cover common networking data types in [Chapter 2](#). There are also many ways of collecting data, and many engineers still do it manually—connecting to devices and systems one by one to gather information. This system works, but it can be improved. A better way is to use automation, as described in this section (and also in [Chapter 2](#)).

Say that you want to collect data on a number of connected devices in your network. For the sake of this example, assume that all devices have either LLDP or CDP configured. One way to solve this task would be to use **show** command output for either LLDP or CDP (depending on the device vendor) to learn the hostnames of the neighbor devices, as shown in [Example 1-1](#).

Example 1-1 **show cdp neighbors** Output on a Cisco Switch

```
Pod6-RCDN6-Border#show cdp neighbors
Capability Codes: R - Router, T - Trans Bridge, B - Source Route B
                  S - Switch, H - Host, I - IGMP, r - Repeater, P -
                  D - Remote, C - CVTA, M - Two-port Mac Relay

Device ID      Local Intrfce     Holdtme   Capability Platform
Switch-172-31-63-161.cisco.com
                  Gig 1/0/13          166        R S I   WS-C3850
Switch-172-31-63-161.cisco.com
                  Gig 1/0/14          149        R S I   WS-C3850
Pod6-SharedSvc.cisco.com
                  Gig 1/0/2           152        R S I   WS-C3650

Total cdp entries displayed : 3
```

You could connect to every device on the network, issue this command, and save the output and later create some document based on the information you gather. However, this activity could take a lot of time, depending on the number of devices on the network. In any case, it would be a boring and time-consuming activity.

With automation, if you have a list of target systems, you can use a tool to connect to the systems, issue commands, and register the output much—all very quickly. Furthermore, the tool could automatically parse the verbose **show** command and then save or display only the required hostnames.

This type of automation is easily achievable and liberates people to focus on tasks that only humans can do.

Note

You will learn later in this book how to collect and parse this type of information from network devices by using tools such as Python and Ansible.

Configuration

Applying point-in-time configurations to devices and systems is another common networking task. Typically, you need the desired configurations and a list of targets to apply the configurations to. The configurations may differ, depending on the target platform, and this can be challenging with a manual process.

Automating configuration activities can greatly improve speed and reduce human error. As in the previous use case, the automation tool connects to the targets, but now, instead of retrieving information, it configures the devices. In addition, you can automate the verification of the configuration.

The following steps show the typical configuration workflow:

Step 1. Connect to a device.

Step 2. Issue precheck **show** commands.

Step 3. Apply the configuration by entering the appropriate commands.

Step 4. Verify the applied configurations by using **show** commands.

Step 5. Issue postcheck **show** commands.

Step 6. Repeat steps 1–5 for all the devices on the list of target platforms.

You can do many kinds of configurations. These are the most common examples of configurations on network devices:

- Shutting down interfaces
- Tuning protocols (based on timers or features, for example)
- Changing interface descriptions
- Adding and removing access lists

In addition to using the simple workflow shown earlier, you can configure more complex workflows, such as these:

- You can configure devices in a specific order.
- You can take actions depending on the success or failure of previous actions.
- You can take time-based actions.
- You can take rollback actions.

In enterprise networking, the tool most commonly used for network automation is Ansible. Some of the reasons for its dominance in this domain are its extensive ecosystem that supports most types of devices, its agentless nature, and its low entry-level knowledge requirements. [Chapter 5](#) illustrates how to access and configure devices using Ansible.

An automation tool may even allow you to define intent (that is, what you want to be configured), and the tool generates the required configuration commands for each platform. This level of abstraction enables more flexibility and loose coupling between components. (Keep this in mind when choosing an automation tool.)

Provisioning

Provisioning is different from configuration, but the two terms are often

confused. *Provisioning*, which occurs before configuration, involves creating or setting up resources. After a resource is provisioned, it is made available and can be *configured*. For example, a user could provision a virtual machine. After that virtual machine boots and is available, the user could configure it with the software required for a particular use case.

As you know, the cloud has become an extension of company networks. In addition to public clouds (such as Amazon Web Services, Google Cloud Platform, and Microsoft Azure), organizations also use private clouds (for example, OpenStack or VMware vCloud suite). In the cloud, provisioning new resources is a daily activity. You can accomplish provisioning by using a graphical user interface, the command line, or application programming interfaces (APIs). Provisioning can also entail creating new on-premises resource such as virtual machines.

As you might have guessed by now, using a graphical interface is significantly slower and error prone than using automation for provisioning. Automating the provisioning of resources is one of the cloud's best practices —for several reasons:

- **Resource dependencies:** Resources may have dependencies, and keeping track of them is much easier for automation than for humans. For example, a virtual machine depends on previously created network interface cards. Network engineers may forget about or disregard such dependencies when doing provisioning activities manually.
- **Cost control:** When you use on-premises resources, you tend to run them all the time. However, you don't always need some resources, and provisioning and deprovisioning resources in the cloud as you need them can be a cost control technique. The cloud allows for this type of agility, as you only pay for what you use.
- **Ease of management:** By automating the provisioning of resources, you know exactly what was provisioned, when, and by whom. Management is much easier when you have this information.

A number of tools facilitate cloud provisioning by defining text-based resource definitions (for example, Terraform and Ansible). See the “Tools” section, later in this chapter, for further details on these tools.

Reporting

Reports are a necessary evil, and they serve many purposes. Generating reports is often seen as a tedious process.

In networking, reports can take many shapes and forms, depending on the industry, technology, and target audience. The basic process for generating a report involves three steps:

Step 1. Gather the required information.

Step 2. Parse the information in the desired format.

Step 3. Generate the report.

This process can vary, but the three main steps always occur.

Network engineers often need to run reports on software versions. Due to vendors' software life cycle policies, companies fairly often need to get an updated view of their software status. This type of report typically consists of the following:

- Devices (platforms and names)
- Current software versions (names and end dates)
- Target software versions (names)

Reports on the hardware life cycle are also common in networking. Vendors support specific hardware platforms for a specific amount of time, and it is important to keep an updated view of the hardware installed base in order to plan possible hardware refreshes.

In addition, configuration best practices and security advisory impact reports are common in networking.

Automation plays a role in reporting. Many tools have automatic report generating capabilities, no matter the format you might need (such as HTML, email, or markdown). The steps required to gather and parse information can be automated just like the steps described earlier for data collection; in fact, reporting features are typically embedded in data collection automation tools. However, there are cases in which you must provide the data input for the report generation. In such cases, you must automate the data gathering and parsing with another tool.

For the example of software versioning mentioned earlier, you could use a tool to connect to network devices to gather the current software version of each device. To get the target versions for the devices, the tool could use vendors' websites or API exposed services. With such information, the automation tool could compile a report just as a human would.

Note

Most of the time, tailoring your own tool is the way to go in reporting. Each company has its own requirements for reports, and it can be difficult to find a tool that meets all of those requirements.

End-to-End Automation

So far, we have described several use cases and scenarios, but all of them have focused on single tasks or components. You know that a network is an ecosystem composed of many different platforms, pieces of vendor equipment, systems, endpoints, and so on. When you need to touch several of these components in order to achieve a goal, you can use end-to-end automation.

Note

End-to-end automation can be complex because different components might have different ways of being accessed, as well as different configurations and metrics. Do not let this put you off; just tread lightly. End-to-end automation is the most rewarding type of automation.

What End-to-End Automation Is

End-to-end automation is not a unique type of automation but a combination of data-driven and task-based automation. It can also be seen as a combination of several use cases of the same type. It is called *end-to-end*

because its goal is to automate a flow from a service perspective. The majority of the time, your needs will fall within this category. Because data-driven and task-based automation are the building blocks for an end-to-end flow, you do need to understand those types as well.

[Figure 1-5](#) shows the topology used in the following example. In this topology, to configure a new Layer 3 VPN service, several components must be touched:

- LAN router or switch
- CPE router
- PE router
- P router

All of these components can be from the same vendor, or they can be from different vendors. In addition, they can have the same software version, or they can have different versions. In each of those components, you must configure the underlay protocol, BGP, route targets, and so on. This would be considered an end-to-end automation scenario.

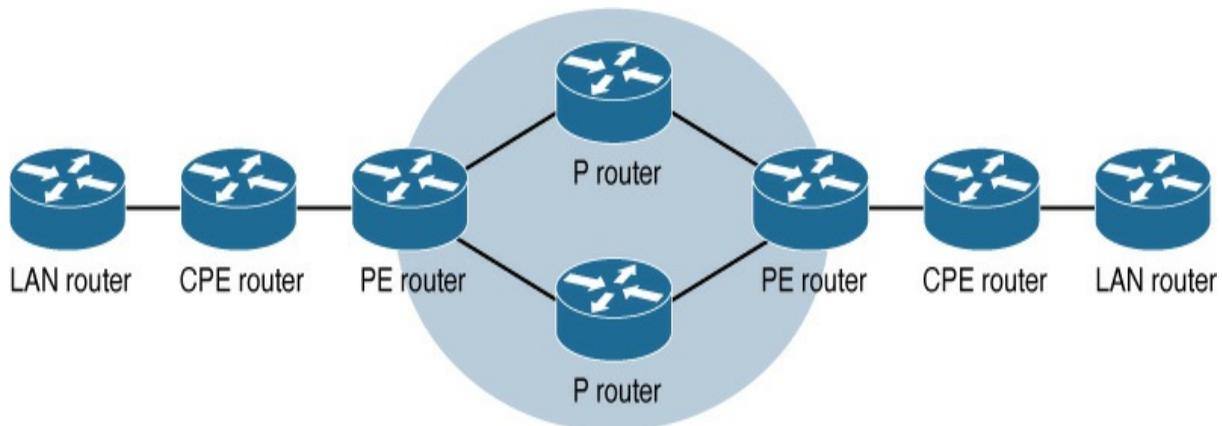


Figure 1-5 Layer 3 VPN Topology

Tip

Think of an end-to-end process you would like to automate. Break it down into smaller data-driven or task-based processes. Your automation job will then be much easier.

End-to-End Automation Use Cases

All of the use cases described in this section are real-life use cases. They were implemented in companies worldwide, at different scales.

Note

All of the use cases discussed so far in this chapter could potentially be applied to an end-to-end scenario (for example, monitoring an entire network, running a compliance report of all devices).

Migration

Migrations—either from an old platform to a new platform or with a hardware refresh—are often time-consuming tasks, and some consider them to be a dreadful activity. They are complex and error prone, and most of the time, migrations impact business operations. Automation can help you make migrations smoother.

Consider the following scenario: You need to migrate end-of-life devices in a data center to new ones. The new ones are already racked and cabled and have management connectivity in a parallel deployment. There is a defined method for carrying out the migration:

Step 1. Collect old device configurations.

Step 2. Create new device configurations.

Step 3. Execute and collect the output of the precheck commands.

Step 4. Configure new devices with updated configurations.

Step 5. Execute and collect the output of postcheck commands.

Step 6. Shift the traffic from the old devices to the new devices.

Step 7. Decommission the old devices.

This is a fairly straightforward migration scenario, but depending on the number of devices and the downtime accepted by the business requirements,

it might be challenging.

An engineer would need to use SSH to reach each old device to collect its configuration, make the required changes in a text editor, use SSH to reach each of the new devices, configure each device, and verify that everything works. Then the engineer can shift the traffic to the new infrastructure by changing routing metrics on other devices and verify that everything is working as expected.

You could automate steps of this process, and, in fact, you have already seen examples of automating these steps:

- Collect the old configurations.
- Generate the new configurations.
- Apply changes on the devices.
- Verify the configurations/services.

Remember that migrations are often tense and prone to errors. Having an automated step-by-step process can mean the difference between a rollback and a green light, and it can help you avoid catastrophic network outages.

Note

A rollback strategy can be embedded into an automated process and automatically triggered if any check, pre or post, is different from what is expected.

Configuration

We covered a configuration use case earlier, when we looked at task-based automation. However, it is important to highlight that end-to-end configuration is just as valuable, if not more so. Automating service configuration involves a collection of automated tasks.

Instead of simply configuring a VLAN on a list of devices, say that you want to run an application on a newly created virtual machine and enable it to communicate. Using the data center topology shown in [Figure 1-6](#), you need to take these steps to achieve the goal:

Step 1. Create a virtual machine (component A).

Step 2. Install the application.

Step 3. Configure Layer 2 networking (component B).

Step 4. Configure Layer 3 networking (components C and D).

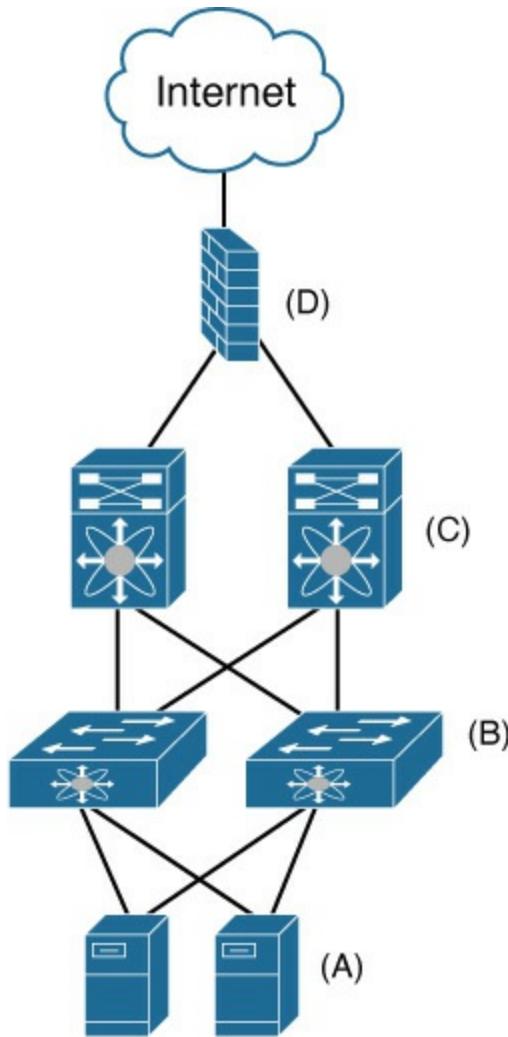


Figure 1-6 Simplified Data Center Topology

These steps are simplified, of course. In a real scenario, you normally have to touch more components to deploy a new virtual machine and enable it to communicate in a separate VLAN. The complexity of the configuration depends on the complexity of the topology. Nonetheless, it is clear that to achieve the simple goal of having an application in a private data center

communicate with the Internet, you require an end-to-end workflow. Achieving this manually would mean touching many components, with possibly different syntax and versions; it would be challenging. You would need to know the correct syntax and where to apply it.

An end-to-end automated workflow can ease the complexity and reduce human errors in this configuration. You would define syntax and other characteristics once, when creating the workflow, and from then on, you could apply it with the press of a button.

The more complex and extensive a workflow is, the more likely it is that a human will make a mistake. End-to-end configurations benefit greatly from automation.

Note

You will see a more in-depth explanation on how to achieve automation in this example in [Chapter 6, “Network DevOps.”](#)

Provisioning

As mentioned earlier, automating provisioning of resources on the cloud is key for taking advantage of cloud technology.

Most companies do not use a single cloud but a combination of services on multiple clouds as well as on-premises devices. Using more than one cloud prevents an organization from being locked to one vendor, helps it survive failures on any of the clouds, and makes it possible to use specialized services from particular vendors.

Provisioning resources across multiple clouds is considered end-to-end automation, especially if you also consider the need for the services to communicate with each other. Provisioning multiple different resources in a single cloud, as with a three-tier application, would also fall within this end-to-end umbrella.

Let’s consider a disaster recovery scenario. Say that a company has its services running on an on-premises data center. However, it is concerned about failing to meet its service-level agreements (SLAs) if anything happens

to the data center. The company needs to keep its costs low.

A possible way of addressing this scenario is to use the cloud and automation. The company could use a provisioning tool to replicate its on-premises architecture to the cloud in case something happens to the data center. This would be considered a cold standby architecture.

The company could also use the same type of tool to scale out its architecture. If the company has an architecture hosting services in a region, and it needs to replicate it to more regions, one possible way to do this is by using a provisioning tool. (A *region* in this case refers to a cloud region, which is cloud provider's presence in a specific geographic region.)

Note

Remember that when we refer to the cloud, it can be a public cloud or a private cloud. Provisioning use cases apply equally to both types.

Testing

Like migration, testing can be an end-to-end composite use case composed of different tasks such as monitoring, configuration, compliance verification, and interaction. Also like migration, testing can be automated. By having automated test suites, you can verify the network status and the characteristics of the network quickly and in a reproducible manner.

Network testing encompasses various categories, including the following:

- Functional testing
- Performance testing
- Disaster recovery testing
- Security testing

Functional testing tries to answer the question “Does it work?” It is the most common type of test in networking. If you have found yourself using **ping** from one device to another, that is a type of functional testing; you use **ping** to verify whether you have connectivity—that is, whether it works. The goal

of more complex types of functional tests may be to verify whether the whole network is fulfilling its purpose end to end. For example, you might use **traceroute** to verify if going from Device A to Device B occurs on the expected network path, or you might use **curl** to verify that your web servers are serving the expected content.

Performance testing is focused on understanding whether something is working to a specific standard. This type of testing is used extensively on WAN links, where operators test the latency of all their service provider links. Another common use of performance testing is to stress a device to make sure it complies with the vendor specifications.

Disaster recovery testing consists of simulating a disaster-like situation, such as a data center outage, to verify that the countermeasures in place occur, and the service is not affected or is minimally affected. This is a more niche use case because you wouldn't test it if you didn't have countermeasures in place (for example, a dual data center architecture).

Finally, security testing is another common type of testing in networking. The goal of this type of testing is to understand the security status of the network. Are the devices secure? Are any open ports? Is everything using strong encryption? Are the devices updated with the latest security patches?

Although all the tests described previously can be done manually—and have been handled that way in the past—moving from manual testing to automated testing increases enterprise agility, enabling faster reactions to any identified issues. Consider a system that periodically verifies connectivity to your critical assets—for example, your payment systems or backend infrastructure—and that measures delay, jitter, and page load times. When you make a network change, such as replacing a distribution switch that breaks, the system can quickly test whether everything is working and allow you to compare the current performance against previously collected data. Performing this task manually would be error prone and time-consuming.

Advanced automated testing systems can trigger remediation actions or alarms.

Tools

There are numerous tools on the market today. The following sections present many of the tools that are worth mentioning in the context of network automation:

- DNA Center
- Cloud event-driven functions
- Terraform
- Ansible
- Chef
- Grafana
- Kibana
- Splunk
- Python

Some of these tools are not fixed tools but rather modular tools that can be tailored and adapted to achieve data-driven network automation goals. This book focuses on such modular tools.

Choosing the right tool for the job is as important in automation as is in other areas. When it comes to choosing a tool, it is important to evaluate your need against what the tool was designed to do. In addition, it is important that the staff responsible for using and maintaining the tool be comfortable with the choice.

Note

For each of the following tools, we highlight in bold the connection to previous illustrated use cases.

DNA Center

DNA Center (DNAC) is a Cisco-developed platform for network management, monitoring, and automation. It allows you to **provision** and **configure** Cisco network devices. It has artificial intelligence (AI) and

machine learning (ML) to proactively **monitor**, **troubleshoot**, and **optimize** the network.

DNAC is software installed in a proprietary appliance, so you cannot install it outside that hardware.

If you are looking to automate a network environment with only Cisco devices, this platform is a quick win. Unfortunately, it does not support all device platforms—not even within Cisco’s portfolio.

In the context of automation, DNAC ingests network data (SNMP, Syslog, NetFlow, streaming telemetry) from devices, parses it, and enables you to quickly complete the following tasks:

- **Set up alerts based on defined thresholds:** You can set up alerts based on metrics pulled from network devices (for example CPU, interface, or memory utilization). Alerts can have distinct formats, such as email or webhooks.
- **Baseline network behaviors:** Baselining involves finding common behavior, such as that a specific switch uplink sees 10 Gigabits of traffic on Monday and only 7 Gigabits on Tuesday. Discovering a good baseline value can be challenging, but baselines are crucial to being able to identify erroneous behaviors. DNAC offers baselining based on ML techniques. You will see more on this topic in [Chapter 3](#).
- **Aggregate several network metrics to get a sense of network health:** Based on a vast amount of captured information, DNAC displays a consolidated view of the status of a network to give a sense of network health. It can quickly provide insights into devices and clients, without requiring an understanding of metrics or logs.
- **Act on outliers:** Outliers can exist for many reasons. An attacker might have taken control of one of your endpoints, or equipment might have broken. Being able to quickly apply preventive measures can make a difference. DNAC allows you to automate what actions are taken in specific situations.
- **Verify compliance status and generate reports:** DNAC enables you to view and export network compliance verification and reports from an appliance as an out-of-the-box functionality.

DNAC is not as modular as some other tools, and enhancing its out-of-the-box functionalities is not supported in most cases.

Cloud Event-Driven Functions

If you have a cloud presence, networking is a part of that. One way of automating in the cloud is to use serverless functions. Cloud functions are event driven, which requires a very specific data-driven automation type. Each cloud vendor has its own implementation, which makes multivendor environments very challenging. Developed code is not portable between different clouds even if they follow the same guidelines.

Event driven means actions are derived from events, or changes of state. Events can be messages, actions, alarms, and so on. An event-driven architecture is typically an asynchronous architecture.

AWS Lambda, Google Cloud Functions, and Azure Functions are examples of this technology.

Say that you want to scale out your architecture (virtual machines or containers) when the CPU utilization of the current instances is high. You can trigger an event when the CPU hits a defined threshold, and the cloud function **provisions** and **configures** new resources to handle the load.

Event-driven functions also shine at **interacting** with many different components. They are typically used to automate the integration between otherwise independent components. For example, when someone tries to connect to a virtual machine but provides invalid credentials, this action can generate an event. The event can then be sent to a cloud function that creates a security alert or a ticket on a network management platform.

Cloud event-driven functions are not limited to interacting with cloud components, although they run on the cloud. They can trigger actions anywhere, including on premises.

Terraform

Terraform is an open-source infrastructure as code (IaC) tool with focus on **provisioning** and management. Terraform uses a declarative model, in which

you define the intended desired state, and Terraform figures out the best way to achieve it; that is, Terraform gathers the current state, compares it to the desired state, and creates a finite set of steps to achieve it. Terraform works best in cloud environments, where representing the infrastructure as code is easier.

Terraform uses the concept of *provider*, which is an abstraction layer for the interaction with the infrastructure. It supports multiple providers, including the following, and the list keeps expanding:

- Amazon Web Services
- Microsoft Azure
- Google Cloud Platform
- IBM Cloud
- VMware vSphere
- Cisco Systems

Terraform is command-line interface (CLI) based, so you just need to install it on a workstation to begin using it. You do not need a dedicated server, although it is common to use dedicated machines in production deployments. Another option is to use HashiCorp's SaaS offering, which is often preferred by large enterprises.

To use Terraform, you define your desired resources in a .tf file. (You will see how to do this later in this section.)

For more complex workflows, your files becomes increasingly complex, you can use modules. A Terraform module is a collection of files that you can import to provide abstraction to otherwise complex or extensive logic.

It is important to note that Terraform stores the state of your infrastructure and refreshes that state before performing any operation on it. That state is used to calculate the difference between what you currently have and what is defined in your desired .tf file.

Using Terraform to create a virtual machine on Amazon Web Services entails the following steps, which are illustrated with examples:

Step 1. Create a .tf configuration file (see [Example 1-2](#)).

Step 2. Initialize Terraform (see [Example 1-3](#)).

Step 3. Optionally verify changes that are to be applied (see [Example 1-4](#)).

Step 4. Apply the changes (see [Example 1-5](#)).

Step 5. Optionally verify whether the applied changes are as expected (see [Example 1-6](#)).

Note

The following examples assume that Terraform is installed, and an AWS account is configured.

Example 1-2 Step 1: Creating a Configuration File

```
$ cat conf.tf
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
    }
  }

  provider "aws" {
    profile = "default"
    region  = "us-west-1"
  }

  resource "aws_instance" "ec2name" {
    ami           = "ami-08d9a394ac1c2994c"
    instance_type = "t2.micro"
  }
}
```

Example 1-3 Step 2: Initializing Terraform

```
$ terraform init
Initializing the backend...

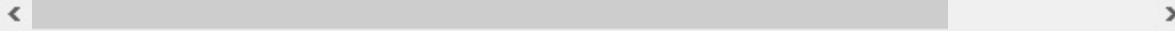
Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v3.22.0...
- Installed hashicorp/aws v3.22.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record th
selections it made above. Include this file in your version contro
so that Terraform can guarantee to make the same selections by def
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform p
any changes that are required for your infrastructure. All Terrafo
should now work.

If you ever set or change modules or backend configuration for Ter
rerun this command to reinitialize your working directory. If you
commands will detect it and remind you to do so if necessary.
```



Example 1-4 Step 3: (Optional) Verifying the Changes That Are to Be Applied

```
$ terraform plan

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# aws_instance.ec2name will be created
+ resource "aws_instance" "ec2name" {
```

```
+ ami = "ami-08d9a394ac1c2994c"
+ arn = (known after apply)
+ associate_public_ip_address = (known after apply)
+ availability_zone = (known after apply)
+ cpu_core_count = (known after apply)
+ cpu_threads_per_core = (known after apply)
+ get_password_data = false
+ host_id = (known after apply)
+ id = (known after apply)
+ instance_state = (known after apply)
+ instance_type = "t2.micro"
# Output omitted #

+ ebs_block_device {
    + device_name = (known after apply)
    + volume_id = (known after apply)
    + volume_size = (known after apply)
    + volume_type = (known after apply)
    # Output omitted #
}

+ metadata_options {
    + http_endpoint = (known after apply)
    + http_put_response_hop_limit = (known after apply)
    + http_tokens = (known after apply)
}

+ network_interface {
    + delete_on_termination = (known after apply)
    + device_index = (known after apply)
    + network_interface_id = (known after apply)
}

+ root_block_device {
    + delete_on_termination = (known after apply)
    + device_name = (known after apply)
    + encrypted = (known after apply)
    + iops = (known after apply)
    + kms_key_id = (known after apply)
    + throughput = (known after apply)
```

```
    + volume_id          = (known after apply)
    + volume_size        = (known after apply)
    + volume_type        = (known after apply)
}
}
```

Plan: 1 to add, 0 to change, 0 to destroy.

Note: You didn't specify an "-out" parameter to save this plan, so can't guarantee that exactly these actions will be performed if "terraform apply" is subsequently run.



Example 1-5 Step 4: Applying the Changes

```
$ terraform apply
```

An execution plan has been generated and is shown below.

Resource actions are indicated with the following symbols:

+ create

Terraform will perform the following actions:

```
# aws_instance.ec2name will be created
+ resource "aws_instance" "ec2name" {
    + ami                      = "ami-08d9a394ac1c2994c"
    + arn                      = (known after apply)
    + associate_public_ip_address = (known after apply)
    + availability_zone         = (known after apply)
    + cpu_core_count            = (known after apply)
    + cpu_threads_per_core      = (known after apply)
    + get_password_data         = false
    + host_id                   = (known after apply)
    + id                        = (known after apply)
    + instance_type              = "t2.micro"
    + source_dest_check          = true
}
```

```
+ subnet_id = (known after apply)
# Output omitted #

+ ebs_block_device {
    + delete_on_termination = (known after apply)
    + device_name = (known after apply)
    + iops = (known after apply)
    + volume_id = (known after apply)
    + volume_size = (known after apply)
    + volume_type = (known after apply)
}

+ metadata_options {
+ http_endpoint = (known after apply)
    + http_put_response_hop_limit = (known after apply)
    + http_tokens = (known after apply)
}

+ network_interface {
    + delete_on_termination = (known after apply)
    + device_index = (known after apply)
    + network_interface_id = (known after apply)
}

+ root_block_device {
    + delete_on_termination = (known after apply)
    + device_name = (known after apply)
    + encrypted = (known after apply)
    + iops = (known after apply)
    + kms_key_id = (known after apply)
    + throughput = (known after apply)
    + volume_id = (known after apply)
    + volume_size = (known after apply)
    + volume_type = (known after apply)
}

}
```

Plan: 1 to add, 0 to change, 0 to destroy.

```
Do you want to perform these actions?  
Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.
```

```
Enter a value: yes
```

```
aws_instance.ec2name: Creating...  
aws_instance.ec2name: Still creating... [10s elapsed]  
aws_instance.ec2name: Creation complete after 19s [id=i-030cd63bd0]  
  
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Example 1-6 Step 5: (Optional) Verifying Whether the Applied Changes Are as Expected

```
$ terraform show  
  
# aws_instance.ec2name:  
resource "aws_instance" "ec2name" {  
    ami                                = "ami-08d9a394ac1c2994c"  
    arn                                = "arn:aws:ec2:us-west-1:05465408  
030cd63bd0a9041d6"  
    associate_public_ip_address      = true  
    availability_zone                 = "us-west-1b"  
    cpu_core_count                    = 1  
    cpu_threads_per_core             = 1  
    disable_api_termination          = false  
    ebs_optimized                     = false  
    get_password_data                = false  
    hibernation                       = false  
    id                                = "i-030cd63bd0a9041d6"  
    instance_state                   = "running"  
    instance_type                     = "t2.micro"  
    ipv6_address_count               = 0  
    ipv6_addresses                   = []  
    monitoring                        = false  
    primary_network_interface_id     = "eni-04a667a8dce095463"
```

```
private_dns          = "ip-172-31-6-154.us-west-1.comp"
private_ip           = "172.31.6.154"
public_dns          = "ec2-54-183-123-73.us-west-1.co"
public_ip            = "54.183.123.73"
secondary_private_ips = []
security_groups     = [
    "default",
]
source_dest_check   = true
subnet_id           = "subnet-5d6aa007"
tenancy              = "default"
volume_tags          = {}
vpc_security_group_ids = [
    "sg-8b17eafdf",
]

credit_specification {
    cpu_credits = "standard"
}

enclave_options {
    enabled = false
}

metadata_options {
    http_endpoint          = "enabled"
    http_put_response_hop_limit = 1
    http_tokens             = "optional"
}

root_block_device {
    delete_on_termination = true
    device_name           = "/dev/xvda"
    encrypted              = false
    iops                  = 100
    throughput             = 0
    volume_id              = "vol-0c0628e825e60d591"
    volume_size             = 8
    volume_type             = "gp2"
```

```
}
```

This example shows that you can provision a virtual machine in three simple required steps: initialize, plan, apply. It is just as easy to provision many virtual machines or different resource types by using Terraform.

Another advantage of using Terraform for provisioning operations is that it is not proprietary to a cloud vendor. Therefore, you can use similar syntax to provision resources across different environments, which is not the case when you use the cloud vendors' provisioning tools. Cloud vendors' tools typically have their own syntax and are not portable between clouds. Furthermore, Terraform supports other solutions that are not specific to the cloud, such as Cisco ACI, Cisco Intersight, and Red Hat OpenShift.

Ansible

Ansible is an open-source **configuration** IaC management tool, although it can be used for a wide variety of actions, such as management, application deployment, and orchestration. Ansible was developed in Python and has its own syntax using the YAML format (see [Example 1-7](#)).

Ansible is agentless, which means you can configure resources without having to install any software agent on them. It commonly accesses resources by using SSH, but other protocols can be configured (for example, Netconf, Telnet). Unlike Terraform, Ansible can use a declarative or a procedural language; in this book, we focus on the declarative language, as it is the recommended approach. Ansible uses a push model.

Ansible can run from a workstation, but it is common to install it in a local server, from which it can more easily reach all the infrastructure it needs to manage.

Note

The following example assumes that Ansible is installed.

Example 1-7 Using Ansible to Configure a New TACACS+ Key

```
$ cat inventory.yml
all:
  children:
    ios:
      hosts:
        switch_1:
          ansible_host: "10.0.0.1"
          ansible_network_os: cisco.ios.ios
  vars:
    ansible_connection: ansible.netcommon.network_cli
    ansible_user: "username"
    ansible_password: "cisco123"

$ cat playbook.yml
---
- name: Ansible Playbook to change TACACS keys for IOS Devices
  hosts: ios
  connection: local
  gather_facts: no

  tasks:
    - name: Gather all IOS facts
      cisco.ios.ios_facts:
        gather_subset: all

    - name: Set the global TACACS+ server key
      cisco.ios.ios_config:
        lines:
        - "tacacs-server key new_key"

$ ansible-playbook playbook.yml -i inventory.yml
PLAY [Ansible Playbook to change TACACS+ keys for IOS Devices]
*****
TASK [Gather all IOS facts]
*****
```

```
ok: [switch_1]

TASK [Set the global TACACS+ server key]
*****
changed: [switch_1]

PLAY RECAP ****
switch_1 : ok=1    changed=1    unreachable=0
failed=0   skipped=0   rescued=0   ignored=0
```

In this example, Ansible gathers device information and changes the TACACS+ key to a new one. There is a single Cisco device in the inventory file, and you can see that the key change succeeds for that specific device.

You could use the facts gathered to determine whether to make the change or not; for example, you might want to execute the change only if a source interface is defined. (You will learn more about such complex playbook workflows in the following chapters.)

Ansible is very extensible. It can also be used to **provision** resources, **interact** with systems, **collect** information, and even perform network **migrations**.

Note

Do not be worried if you don't yet fully understand [Example 1-7](#). [Chapters 4](#) and [5](#) describe Ansible's installation and concepts (including playbooks, inventory, and roles), as well as how to achieve common networking tasks.

Chef

Chef is a **configuration** management tool written in Ruby. It allows you to define IaC but, unlike the tools described so far, uses a pull model. Chef uses a procedural language.

Chef configurations are described in *recipes* (see [Example 1-8](#)). Collections

of recipes are stored in *cookbooks*.

Typically, a cookbook represents a single task (for example, configuring a web server). You can use a community cookbook instead of building your own; they cookbooks cover most of the common tasks.

Chef's architecture consists of three components (see [Figure 1-7](#)):

- **Nodes:** A node represents a managed system. Nodes can be of multiple types and are the ultimate targets you want to configure.
- **Chef server:** The Chef server provides a centralized place to store cookbooks and their respective recipes, along with metadata for the registered nodes. Nodes and the Chef server interact directly.
- **Workstation:** A user uses a workstation to interact with Chef. A workstation is also where you develop and test cookbooks.

Chef is targeted at managing systems as it requires the installation of an agent. Nonetheless, it can manage infrastructure that allows the installation of the agent (for example, Cisco Nexus 9000).

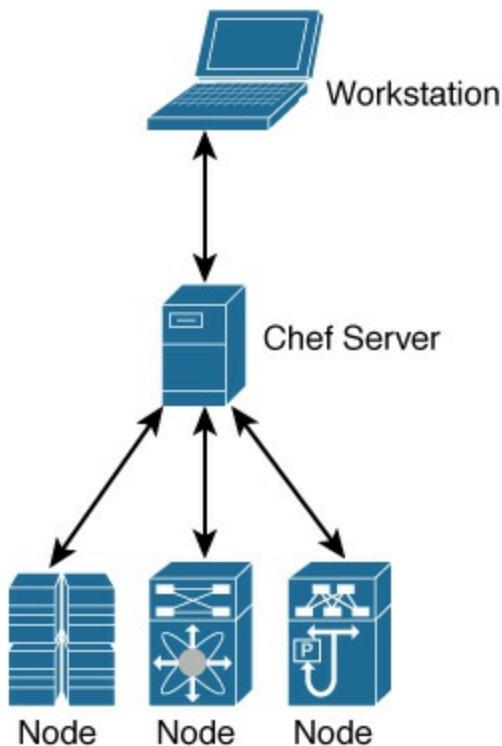


Figure 1-7 Chef Architecture

Note

The following example assumes that the Chef architecture is in place.

Example 1-8 *Chef Recipe to Configure New Web Servers, Depending on Operating System Platform*

```
package 'Install webserver' do
  case node[:platform]
  when 'redhat', 'centos'
    package_name 'httpd'
    action :install
  when 'ubuntu', 'debian'
    package_name 'apache2'
    action :install
  end
end
```

Chef managed systems (nodes) are constantly monitored to ensure that they are in the desired state. This helps avoid configuration drift.

Kibana

Kibana is an open-source tool for data visualization and **monitoring**. Kibana is simple to install and to use.

Kibana is part of the ELK stack, which is a collection of the tools Elasticsearch, Logstash, and Kibana. Logstash extracts information from collected logs (network or application logs) from different sources and stores them on Elasticsearch. Kibana is the visualization engine for the stack. A fourth common component is Beats, which is a data shipper. Beats provides a way of exporting data from any type of system to Logstash, using what an *agent*. Agents in the infrastructure capture data and ship it to Logstash.

It is a common misconception that Kibana collects data directly from a

network—but it doesn’t. It uses the other stack members to do the collection.

You can use Kibana to set up dashboards and alerts and to do analytics based on parsing rules that you define (see [Figure 1-8](#)). It is a tool focused on *log data* processing.

Kibana offers powerful visualization tools, such as histograms and pie charts, but it is by far most used in integration with Elasticsearch as a visualization and query engine for collected logs.

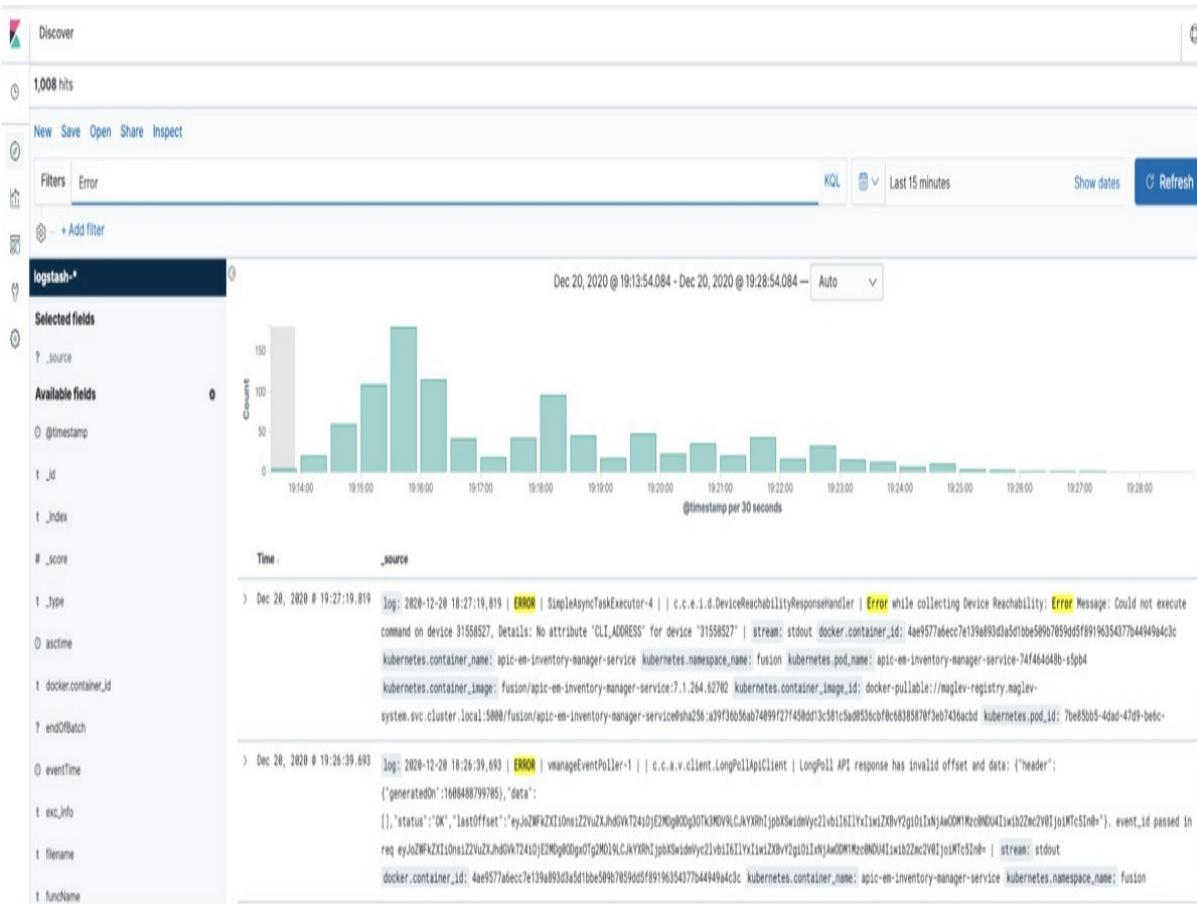


Figure 1-8 Kibana Search Dashboard

Grafana

Grafana is an open-source tool for data visualization, analytics, and **monitoring**. It can be deployed either on-premises or managed in the cloud; there is also a SaaS offering available. Whereas Kibana focuses on log data, Grafana is focused on *metrics*. You can use it to set up dashboards, alerts, and

graphs from network data. You might use it, for example, to visualize the following:

- CPU utilization
- Memory utilization
- I/O operations per second
- Remaining disk space

[Figure 1-9](#) shows an example of a Grafana dashboard that shows these metrics and more.



Figure 1-9 *Grafana Dashboard*

Grafana does not store or collect data. It receives data from other sources where these types of metrics are stored—typically time series databases (for example, InfluxDB). However, it can also integrate with other database types (such as PostgreSQL) and cloud monitoring systems (such as AWS

CloudWatch). In order for the data to be populated in these databases, you must use other tools, such as SNMP.

Because it is able to pull data from many sources, Grafana can provide a consolidated view that other tools fail to provide.

Some may not consider Grafana a network automation solution, but its monitoring capabilities, which you can use to track errors or application and equipment behaviors, along with its alerting capabilities, earn it a spot on this list.

Splunk

Splunk is a distributed system that aggregates, parses, and analyses data. Like Grafana, it focuses on **monitoring**, alerting, and data visualization. There are three different Splunk offerings: a free version, an enterprise version, and a SaaS cloud version.

Although known in the networking industry for its security capabilities, Splunk is on this list of automation tools due to its data ingestion capabilities. It can ingest many different types of data, including the following networking-related types:

- Windows data (registry, event log, and filesystem data)
- Linux data (Syslog data and statistics)
- SNMP
- Syslog
- NetFlow/IPFIX
- Application logs

The Splunk architecture consists of three components:

- Splunk forwarder
- Splunk indexer
- Splunk search head

The Splunk forwarder is a software agent that is installed on endpoints to collect and forward logs to the Splunk indexer. The Splunk forwarder is

needed only when the endpoint that is producing the logs does not send them automatically, as in the case of an application. In the case of a router, you could point the Syslog destination directly at the indexer, bypassing the need for a forwarder.

There are two types of Splunk forwarders:

- **Universal forwarder:** The most commonly used type is the universal forwarder, which is more lightweight than its counterpart. Universal forwarders do not do any preprocessing but send the data as they collect it (that is, raw data). This results in more transmitted data.
- **Heavy forwarder:** Heavy forwarders perform parsing and send only indexed data to the indexer. They reduce transmitted data and decentralize the processing. However, this type of forwarder requires host machines to have processing capabilities.

The Splunk indexer transforms collected data into events and stores them. The transformations can entail many different operations, such as applying timestamps, adding source information, or doing user-defined operations (for example, filtering unwanted logs). The Splunk indexer works with universal forwarders; with heavy forwarders, the Splunk indexer only stores the events and does not perform any transformation. You can use multiple Splunk indexers to improve ingestion and transformation performance.

The Splunk search head provides a GUI, a CLI, and an API for users to search and query for specific information. In the case of a distributed architecture with multiple indexers, the search head queries several indexers and aggregates the results before displaying the results back to the user. In addition, you can define alerts based on received data and generate **reports**.

[Figure 1-10](#) shows an example of a Splunk dashboard that shows software variations among a firewall installation base along with the number of devices and their respective logical context configurations.

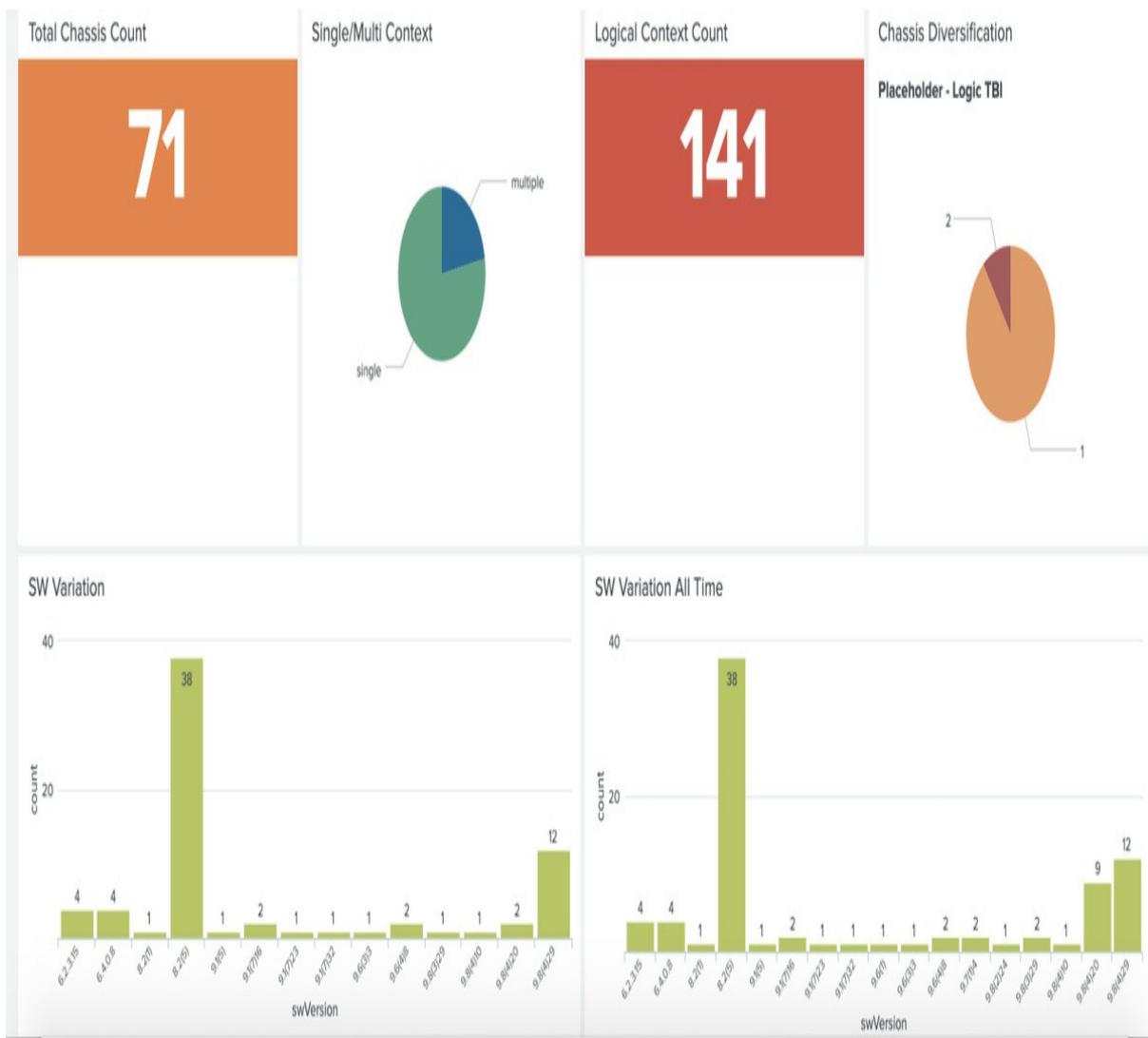


Figure 1-10 Splunk Firewall Information Dashboard

Python

Although Python is not an automation tool per se, it provides the building blocks to make an automation tool. Many of the automation tools described in this book are actually built using Python (for example, Ansible).

Other programming languages could also be used for automation, but Python has a lot of prebuilt libraries that make it a perfect candidate. The following are some examples of Python libraries used to automate networking tasks:

- Selenium

- Openpyxl
- Pyautogui
- Paramiko
- Netmiko

The main advantage of Python is that it can work with any device type, any platform, any vendor, and any system with any version. However, its advantage is also its drawback: The many possible customizations can make it complex and hard to maintain.

When using Python, you are actually building a tool to some extent instead of using a prebuilt one. This can make it rather troublesome.

Example 1-9 defines a device with all its connection properties, such as credentials and ports (as you could have multiple devices). This example uses the Netmiko Python module to connect to the defined device and issues the **show ip int brief** command, followed by the Cisco CLI commands to create a new interface VLAN 10 with the IP address 10.10.10.1. Finally, the example issues the same command as in the beginning to verify that the change was applied.

You can see that this example defines the device type. However, the Python library supports a huge variety of platforms. In the event that your platform is not supported, you could use another library and still use Python. This is an example of the flexibility mentioned earlier. A similar principle applies to the commands issued, as you can replace the ones shown in this example with any desired commands.

Example 1-9 Using Python to Configure and Verify the Creation of a New Interface VLAN

```
from netmiko import ConnectHandler

device = {
    "device_type": "cisco_ios",
    "ip": "10.201.23.176",
    "username": "admin",
    "password": "cisco123",
```

```
"port" : 22,  
"secret": "cisco123"  
}  
  
ssh_connect = ConnectHandler (**device)  
  
print("BEFORE")  
result = ssh_connect.send_command("show ip int brief")  
print(result)  
  
configcmds=["interface vlan 10", "ip add 10.10.10.1 255.255.255.0"]  
ssh_connect.send_config_set(configcmds)  
  
print("AFTER")  
result = ssh_connect.send_command("show ip int brief")  
ssh_connect.disconnect()  
print("")  
print(result)
```

Figure 1-11 shows a sample execution of the script in [Example 1-9](#).

```

ivpinto@IVPINTO-M-K16Y Desktop % python device.py
BEFORE
Vlan1           11.1.1.3      YES NVRAM  up
Vlan100         10.100.10.1   YES NVRAM  up
Vlan112         unassigned    YES unset   up
Vlan113         unassigned    YES unset   up
Vlan114         unassigned    YES unset   up
AFTER
Vlan1           11.1.1.3      YES NVRAM  up
Vlan10          10.10.10.1   YES manual  down
Vlan100         10.100.10.1  YES NVRAM  up
Vlan112         unassigned    YES unset   up
Vlan113         unassigned    YES unset   up
Vlan114         unassigned    YES unset   up

```

Figure 1-11 Sample Python Script Execution to Change an Interface IP Address

Summary

We have covered three types of automation in this chapter:

- **Data-driven automation:** Actions triggered by data
- **Task-based automation:** Manually triggered tasks
- **End-to-end automation:** A combination of the other two types

This chapter describes a number of use cases to spark your imagination about what you might want to automate in your company. Now you know several situations where automation can enable you to achieve better results.

This chapter also takes a look at several automation tools. It describes what they are and in which use cases they shine. In later chapters we will dive deeper into some of these tools.

Review Questions

You can find answers to these questions in [Appendix A, “Answers to Review Questions.”](#)

- 1.** You must append a prefix to the hostname of each of your devices, based on the device’s physical location. You can determine the location based on its management IP address. What type of automation is this?

 - a.** Data-driven
 - b.** Task-based
 - c.** End-to-end
- 2.** What use case is described in the question 1 scenario?

 - a.** Optimization
 - b.** Configuration
 - c.** Data collection
 - d.** Migration
- 3.** Lately, your network has been underperforming in terms of speed, and it has been bandwidth constrained. You decide to monitor a subset of key devices that you think are responsible for this behavior. In this scenario, which model is more suitable for data collection?

 - a.** Push model
 - b.** Pull model
- 4.** Which model does SNMP traps use?

 - a.** Push model
 - b.** Pull model
- 5.** Describe *configuration drift* and how you can address it in the context of network devices.

6. Which of these tools can help you to achieve a configuration use case in an environment with equipment from multiple vendors and different software versions?

- a.** Splunk
- b.** Ansible
- c.** DNA Center
- d.** Grafana
- e.** Kibana
- f.** Terraform

7. The number of users accessing your web applications has grown exponentially in the past couple months, and your on-premises infrastructure cannot keep up with the demand. Your company is considering adopting cloud services. Which of the following tools can help with automated provisioning of resources in the cloud, including networking components? (Choose two.)

- a.** Splunk
- b.** Cloud event-driven functions
- c.** DNA Center
- d.** Terraform
- e.** Python

8. The number of servers in your company has kept growing in the past 5 years. Currently, you have 1000 servers, and it has become impossible to manually manage them. Which of the following tools could you adopt to help manage your servers? (Choose two.)

- a.** Kibana
- b.** Terraform
- c.** DNA Center

d. Chef

e. Ansible

9. Your company has not been using the log data from devices except when an issue comes up, and someone needs to investigate the logs during the troubleshooting sessions. There is no centralized logging, even when performing troubleshooting, and engineers must connect to each device individually to view the logs. Which of the following tools could address this situation by centralizing the log data and providing dashboards and proactive alerts? (Choose two.)

a. Kibana

b. Grafana

c. Splunk

d. DNA Center

10. You are using Terraform to provision your cloud infrastructure in AWS. Which of the following is a valid step but is not a required?

a. Initialize

b. Plan

c. Apply

d. Create

11. You are tasked with automating the configuration of several network devices. Due to the very strict policy rules of your company's security department, you may not install any agent on the target devices. Which of the following tools could help you achieve automation in this case?

a. Chef

b. Terraform

c. Ansible

d. DNA Center

12. True or false: Kibana excels at building visualization dashboards and setting up alerts based on metric data.

a. True

b. False

13. True or false: You can use cloud event-driven functions to interact with on-premises components.

a. True

b. False

Chapter 2. Data for Network Automation

Most networks, like snowflakes, are unique. This makes gathering data from a particular network a unique task. You saw in [Chapter 1, “Types of Network Automation,”](#) that data plays an important role in automation, and in this chapter we explore how important data really is.

This chapter discusses the most common data formats and models that you will interact with in a networking environment:

- YAML
- XML
- JSON
- Syslog
- NetFlow
- IPFIX
- YANG

This chapter also covers how to collect data from a network. It specifically focuses on the relevant methods used today, such as APIs, model-driven techniques, and log exporters.

The Importance of Data

Data is the most important asset of the 21st century. Companies like Google, Facebook, and Instagram profit from data; they are, you might say, in the data business. These are currently among the most valuable companies of all time.

Data can allow you to understand and solve problems, review performance,

improve processes, and make better decisions—as long as you can derive insights from it. The step between data and insight is where the value lies.

Of course, data plays a role in network automation. You can automate actions based on insights, or you can gather network data and try to predict consequences. Data can be the difference between a migration being a success or a failure as it can enable you to quickly identify a drift between the deployed configuration and the expected configuration. Data can also help you anticipate and identify a defect on power supplies that disrupts the network by continuously comparing measurements. And data from multiple logs can help you determine what has been crashing a switch.

Data Formats and Models

Data can come in many formats. It is important to know at least the common ones in order to be able to correctly interpret them. Without this understanding, misinterpretation is possible. Take, for example, the word *moron*. In English, it does not have a very nice meaning, but in Welsh, it means carrot. This is an oversimplified example, but it illustrates the type of misinterpretation that can occur with network data.

Data formats play an important role when interacting with systems. Some components (such as, APIs) or tools (such as, Ansible), as you will see later in this book, work with specific data formats. You need to know how to compose data in such formats to be able to take advantage of these components and tools.

Data models play an important role but are often confused with data formats. A data model defines what values and types data can exist on, as well as how each entity relates to each other entity. In short, the data model defines the data structure. It abstracts the underlying configuration and exposes it in a common format.

To better understand the difference between a data format and a data model, say that you are filling a form on a website. A data format would be the representation of words in letters, based on UTF-8 or ASCII, for example, whereas a data model would restrict the types of words you can insert in each field of the form (for example, requiring @ and .com in the email field).

Data models are very important, as you will see in this chapter, because systems want to enforce a specific data structure and ranges of values—not simply data in a specific format. For example, a router wants the OSPF process ID to be an integer ranging from 1 to 65535, not any integer.

YAML

YAML, which initially stood for *Yet Another Markup Language* and now stands for *YAML Ain't Markup Language*, is a language designed with humans in mind. Its format is human friendly, and it is used with tools such as Ansible. YAML has a number of quirks, including the following:

- A YAML file should start with three dashes (---), although this is not mandatory.
- YAML is case sensitive.
- A YAML file has the extension .yaml or .yml.
- Indentation is important in YAML, unlike in other data formats, such as XML. In a file, the number of spaces used for indentation must always be the same; for example, if you start by using two spaces, you must use two spaces in the rest of the file. By convention, two spaces are typically used, and tabs are not permitted.

[Example 2-1](#) shows an example of a YAML file with correct indentation and an example of a YAML file with invalid indentation.

Example 2-1 YAML File Example

```
Valid YAML file:  
---  
devices:  
    csr1000v:  
        address: "10.0.0.1"  
        os: "iosxe"  
    nexus7000:  
        address: "10.0.0.2"  
        os: "nxos"  
    nexus9000:
```

```

address: "10.0.0.3"
os: "nxos"

Invalid YAML file:

---
devices:
  csr1000v:
    address: "10.0.0.1"
    os: "iosxe"
  nexus7000:
    address: "10.0.0.2"
    os: "nxos"
  nexus9000:
    address: "10.0.0.3"
    os: "nxos"

```

In [Example 2-1](#), the valid file has multiple elements, and each child element has two spaces more than its parent; for example, the element *csr1000v* starts two spaces after from its parent *devices*. In the invalid file, you can see that the element *address* has four spaces more than its parent *nexus9000*, although the element *os* under the same parent has only two spaces.

In YAML there are three data types (see [Example 2-2](#)):

- **Scalar:** A simple value for a single key
- **List:** A collection of values for a single key
- **Dictionary:** A collection of key/value pairs for a single key

YAML also supports nested data types.

It is important to represent data in the way that best suits your purpose. Doing so makes it easier to access the data when it is stored.

Example 2-2 YAML Data Types

```

---
scalar_N9K: "10.0.0.3"
devices_list:
  - "CSR1000v"

```

```
- "N7K"
devices_dictionary:
  CSR1000v:
    address: "10.0.0.1"
    os: "iosxe"
  N7K:
    address: "10.0.0.2"
    os: "nxos"
```

YAML supports flow-style data. All of the previous examples are in block style. [Example 2-3](#) shows a flow style example.

Note

Even though YAML supports flow-style data, you should use block style whenever possible as it makes reading the code much easier.

Example 2-3 Flow-Style YAML

```
---
scalar_N9K: "10.0.0.3"
devices_list: ["CSR1000v", "N7K"]
devices_dictionary: {CSR1000v: {address: "10.0.0.1", os: "iosxe"}, N7K: {address: "10.0.0.2", os: "nxos"}}
```

In [Example 2-3](#), you can see that in flow style, lists are represented using the square brackets notation, and dictionaries are represented using curly braces. The scalar data type is represented the same way as in block style.

Sometimes you might want to add extra information to a YAML file—for example, to explain complex functions or to explain your reasoning when you share files with a broader team. You can achieve this by using the # symbol, which starts a comment in YAML. When a YAML file is parsed, comments are ignored. The following example uses a comment to indicate that a specific device might change IP address in the future:

```
---
```

```
devices:  
  csr1000v:  
    address: "10.0.0.1" #this device might change IP address  
    os: "iosxe"
```

Say that you would like to enforce a specific structure for data. For example, you might want each entry to be a pair (device name and IP address), as shown here:

```
---
```

```
device01: "10.0.0.1"  
device02: "10.0.0.2"  
device03: "10.0.0.3"
```

In the following YAML snippet, you would likely run into trouble because device01 and device02 have different data types than you expect:

```
---
```

```
device01:  
  - "10.0.0.1"  
device02: True  
device03: "10.0.0.3"
```

Instead of being string scalars, device01 is a list, and device02 is scalar with a Boolean value. From a data format perspective, this is a well formatted file, but the data types are problematic.

The data enforcement in this example is typically called a *data model* (or, sometimes, a *schema*). To achieve this type of enforcement in YAML you would have to use a tool as there are no native methods available. There are, however, plenty of tools available. For example, the following code would be the schema for the previous example using the Yamale tool:

```
---
```

```
device01: str()  
device02: str()  
device03: str()
```

This Yamale schema enforces value types so that each device key must be a string. If you used this schema with the previous example, where devices

have values of type Boolean and list, even though that example is a well-formed YAML, the data would not be accepted.

Using data models is critical for data validation and security. You will see in the following sections that different data formats have different ways of ensuring that received data has the expected structure.

Note

You will see many more YAML examples and also use YAML yourself in [Chapter 5, “Using Ansible for Network Automation,”](#) and [Chapter 6, “Network DevOps.”](#)

XML

XML, which stands for *Extensible Markup Language*, was designed with storage and transportation of data in mind. XML was designed to be both human and machine readable—although it is not as human readable as YAML.

XML is often used in machine-to-machine interactions. It is software and hardware independent. In [Example 2-4](#), you can see that XML resembles HTTP, as it has a hierarchical format. Unlike HTTP tags, however, XML tags are not predefined but are authored by the creator of an XML file.

Example 2-4 XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<interfaces>
    <interface type="vlan" number="90">
        <address>10.90.0.1</address>
        <mask>24</mask>
        <description>staging</description>
        <name>Vlan90</name>
        <speed>1000</speed>
        <status>up</status>
    </interface>
```

```
<interface type="vlan" number="100">
    <address>10.100.0.1</address>
    <mask>24</mask>
    <description>production</description>
    <name>Vlan100</name>
    <speed>1000</speed>
    <status>up</status>
</interface>
<interface type="vlan" number="110">
    <address>10.110.0.1</address>
    <mask>24</mask>
    <description>test</description>
    <name>Vlan110</name>
    <speed>1000</speed>
    <status>up</status>
</interface>
</interfaces>
```

Note

The first line of this XML document indicates the XML version and data encoding being used: `<?xml version="1.0" encoding="UTF-8"?>`. Encoding is the process of converting characters to their binary equivalent—in this case, UTF-8.

XML documents are formed as trees. An XML tree starts at a top element, called the *root* element. All elements can have *child* elements (see [Example 2-5](#)). The relationships between elements can be described as follows:

- **Parent:** An element one level above in the hierarchy
- **Child:** An element one level below in the hierarchy
- **Sibling:** An element on the same level in the hierarchy

Example 2-5 XML Root Hierarchy

```
<Root>
```

```
<ChildOfRoot>
    <Sibling1></Sibling1>
    <Sibling2></Sibling2>
</ChildOfRoot>
</Root>
```

XML elements are formed using case-sensitive tags that occur in pairs. There must be an opening tag, which starts with <, and a closing tag, which starts with </.

Each XML element can contain data between the opening and closing tags. In [Example 2-6](#) you can see that the IP address 10.0.0.1 is associated with the address tag, and you can also see a description.

Example 2-6 *XML Element Data*

```
<Interface>
    <address>10.0.0.1</address>
    <description>test interface</description>
</Interface>
```

In XML, spacing is not important because the parser is guided by the tags. For example, the XML document in [Example 2-7](#) would yield to the same result as [Example 2-6](#).

Example 2-7 *XML Element Data Without Spaces*

```
<Interface>
<address>10.0.0.1</address>
<description>test interface</description>
</Interface>
```

XML tags can have attributes that indicate properties. As shown in [Example 2-8](#), an interface can have multiple types. The first interface in this example is of type *vlan* and has the number *90*, and the second interface is a *physical* interface numbered *1/1*. Attributes must always be quoted.

Example 2-8 XML Document with Interface Attributes

```
<Interfaces>
    <Interface type="vlan" number="90">
        <address>10.0.0.1</address>
        <description>This is an SVI</description>
    </Interface>
    <Interface type="physical" number="1/1">
        <address>10.100.0.1</address>
        <description>This is a physical interface</description>
    </Interface>
</Interfaces>
```

As previously mentioned, XML tags are not predefined but authored by the creator of a document. When you add together XML documents from different sources, it is possible for names to collide because they use the same tags to refer to different objects (see [Example 2-9](#)). The solution to this issue is to use XML namespaces.

An XML namespace is a collection of XML elements identified by a resource identifier. This identifier provides a way of distinguishing between elements of the same name, thus avoiding naming conflicts.

Example 2-9 XML Name Collision

```
Network Interface
<Interface type="vlan" number="90">
    <address>10.0.0.1</address>
    <description>This is an SVI</description>
</Interface>
Communication Interface
<Interface>
    <protocol>gRPC</protocol>
    <address>192.168.0.1</address>
</Interface>
```

You can define namespaces explicitly or implicitly. When you define a

namespace explicitly, you must prefix every element. In [Example 2-10](#), you can see that *address* and *description* are prefixed with *if*.

Example 2-10 XML Explicit Namespace Definition

```
<if:Interface xmlns:if="urn:example.com:InterfaceInfo">
    <if:address>10.0.0.1</if:address>
    <if:description>This is an SVI</if:description>
</if:Interface>
```

If all the elements in an XML document refer to the same namespace, you can simply use the implicit format shown in [Example 2-11](#). In this case, prefixing is no longer necessary.

Example 2-11 XML Implicit Namespace Definition

```
<Interface xmlns="urn:example.com:InterfaceInfo">
    <address>10.0.0.1</address>
    <description>This is an SVI</description>
</Interface>
```

We have looked at how an XML document is formatted, but we have not talked about how to verify that a document is well formed according to a specification. This is where *document type definitions (DTDs)* and *schemas* (with *XML Schema Definition [XSD]*) come in.

XML schemas are preferred over DTDs because they are written in XML, they support data types, and they are extensible. Therefore, we don't really talk about DTDs here.

Schemas help independent people agree on a data format and verify XML data, and they can be sent along with XML files as format descriptors. Schemas provide a data modeling language for XML. [Example 2-12](#) shows the format of an XML schema.

The XML document in [Example 2-12](#) (Interfaces.xml) has two attributes on the Interfaces element. The first attribute, which starts with *xmlns:xsi*, informs the parser that this XML should be validated against a schema. The

second attribute tells the parser where to find the schema. The schema starts with the `xs:schema` tag, using the standard URI followed by the document schema. The first element is *Interfaces*, which is a complex type because it contains other elements. It is followed by *Interface*, which is also a complex type with child elements.

Example 2-12 XML Document and Corresponding Schema

```
XML document - Interfaces.xml

<?xml version="1.0" encoding="UTF-8"?>
<Interfaces xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Example.xsd">
    <Interface type="vlan">
        <address>10.0.0.1</address>
        <description>This is an SVI</description>
    </Interface>
</Interfaces>

XML schema - Example.xsd

<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="Interfaces">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Interface">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="address" type="xs:string"/>
                        <xs:element name="description" type="xs:string"/>
                    </xs:sequence>
                    <xs:attribute name="type" type="xs:string" use="required">
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```



Upon receiving an XML document, a common task is to extract specific information from it. There are many ways of doing this, and two important tools to have in mind are XPath and XQuery. *XPath* is a query language for XML documents that makes gathering specific fields easy. *XQuery* is a functional programming language that can be used to extract or manipulate data from XML documents; it is sometimes seen as a superset of XPath.

Examine the XML document in [Example 2-13](#).

Example 2-13 XML Document for Interfaces on a Network Device

```
<Interfaces>
    <Interface type="vlan" number="90">
        <address>10.90.0.1</address>
    </Interface>
    <Interface type="vlan" number="100">
        <address>10.100.0.1</address>
    </Interface>
    <Interface type="vlan" number="110">
        <address>10.110.0.1</address>
    </Interface>
    <Interface type="physical" number="2/2">
        <address>10.2.2.1</address>
    </Interface>
    <Interface type="physical" number="1/1">
        <address>10.1.1.1</address>
    </Interface>
</Interfaces>
```

With this example, say that you would like to retrieve only the IP addresses. The following XPath expression would allow you to do this:

```
//address
```

You would get the following result from running this query:

```
<address>10.90.0.1</address>
```

```
<address>10.100.0.1</address>
<address>10.110.0.1</address>
<address>10.2.2.1</address>
<address>10.1.1.1</address>
```

In the case, you would only need the IP address, so you could instead run the following query:

```
//address/text()
```

You would get the following result from running this query:

```
'10.90.0.1'
'10.100.0.1'
'10.110.0.1'
'10.2.2.1'
'10.1.1.1'
```

Note

An in-depth discussion of XPath and XQuery is beyond the scope of this book. However, you should keep in mind that they are very useful tools for extracting information from XML documents.

JSON

JSON, which is short for JavaScript Object Notation, is basically a lightweight version of XML. It is easy to read for both machines and humans, and it is therefore more common than XML in human-to-machine interactions and just as common as XML in machine-to-machine interactions. Like XML, JSON has a hierarchical structure.

It is possible to convert XML files to JSON and vice versa. [Example 2-14](#) shows an example of a conversion. There are plenty of tools that do such conversions for you.

Example 2-14 XML-to-JSON Conversion

XML Document

```
<Interfaces>
  <Interface type="vlan" number="90">
    <address>10.90.0.1</address>
    <mask>24</mask>
    <description>staging</description>
    <name>Vlan90</name>
    <speed>1000</speed>
    <status>up</status>
  </Interface>
  <Interface type="vlan" number="100">
    <address>10.100.0.1</address>
    <mask>24</mask>
    <description>production</description>
    <name>Vlan100</name>
    <speed>1000</speed>
    <status>up</status>
  </Interface>
  <Interface type="vlan" number="110">
    <address>10.110.0.1</address>
    <mask>24</mask>
    <description>test</description>
    <name>Vlan110</name>
    <speed>1000</speed>
    <status>up</status>
  </Interface>
</Interfaces>
JSON Equivalent
{
  "Interfaces": [
    {
      "address": "10.90.0.1",
      "mask": "24",
      "description": "staging",
      "name": "Vlan90",
      "speed": "1000",
      "status": "up",
      "_type": "vlan",
      "_number": "90"
    },
  ]
```

```

{
    "address": "10.100.0.1",
    "mask": "24",
    "description": "production",
    "name": "Vlan100",
    "speed": "1000",
    "status": "up",
    "_type": "vlan",
    "_number": "100"
},
{
    "address": "10.110.0.1",
    "mask": "24",
    "description": "test",
    "name": "Vlan110",
    "speed": "1000",
    "status": "up",
    "_type": "vlan",
    "_number": "110"
}
]
}

```

A JSON file has the extension .json. Inside it are *values* separated by commas. Values can be the following types:

- Strings
- Numbers
- Booleans (true or false)
- Null (an empty value)
- Arrays (ordered lists of values, denoted by square brackets)
- Objects (unordered collections of key/value pairs, denoted by curly braces)

[Example 2-15](#) shows what each of these types looks like. In this example, *Interface* is an object with *name* as a string, *speed* as a number, an array of *IP*

addresses, a Boolean to record *enabled* status, and *link_utilization* with a null value because the interface is currently in the down state.

Unlike YAML and XML, JSON does not allow comments in a file.

Example 2-15 JSON Types

```
{  
    "Interface": {  
        "name": "GigabitEthernet1/1",  
        "speed": 1000,  
        "ip_addresses": [  
            "10.0.0.1",  
            "10.0.0.1"  
        ],  
        "enabled": false,  
        "link_utilization": null  
    }  
}
```

Just like XML or YAML, JSON can include data model validations. Using a JSON schema is one of the ways to enforce a data model. A JSON schema, which is written in JSON, is in a declarative format and describes the structure that data should have. A JSON schema is used for the same purpose as schemas in other data formats: to ensure the validity of data. [Example 2-16](#) shows a JSON schema called the Ansible Inventory schema.

Example 2-16 Ansible Inventory JSON Schema

```
{  
    "$schema": "http://json-schema.org/draft-04/schema#",  
    "$id": "https://json.schemastore.org/ansible-inventory",  
    "title": "Ansible Inventory",  
    "description": "Root schema of Ansible Inventory",  
    "type": "object",  
    "definitions": {  
        "group": {  
            "type": "object",  
            "properties": {  
                "hosts": {  
                    "type": "array",  
                    "items": {  
                        "type": "object",  
                        "properties": {  
                            "name": {  
                                "type": "string",  
                                "format": "hostname",  
                                "description": "Name of the host",  
                                "required": true  
                            },  
                            "vars": {  
                                "type": "object",  
                                "properties": {  
                                    "all": {  
                                        "type": "object",  
                                        "properties": {  
                                            "type": {  
                                                "type": "string",  
                                                "enum": ["host", "group"],  
                                                "description": "Type of object",  
                                                "required": true  
                                            },  
                                            "value": {  
                                                "type": "anytype",  
                                                "description": "Value of the host or group",  
                                                "required": true  
                                            }  
                                        }  
                                    }  
                                }  
                            }  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

```
"properties": {
    "hosts": {
        "type": ["object", "string"],
        "patternProperties": {
            "[a-zA-Z._0-9)": {
                "type": ["object", "null"]
            }
        }
    },
    "vars": {
        "type": "object"
    },
    "children": {
        "patternProperties": {
            "[a-zA-Z_0-9)": {
                "$ref": "#/definitions/group"
            }
        }
    }
},
"properties": {
    "all": {
        "$ref": "#/definitions/group"
    }
}
}
```

Note

Although Ansible can use JSON inventory files, the common data format used in the industry for this purpose is YAML, due to human readability.

You typically use JSON to retrieve and send data using APIs, as described later in this chapter.

Syslog

Syslog is a network-based logging protocol that is incredibly easy to use as a trigger for actions. If you deal with networks on the daily basis, you know what it is, and you can use this section as a memory refresher.

Syslog functions on a push model with a typical centralized architecture. Syslog messages are sent to a Syslog server for ease of analysis and correlation. The number of messages sent depends on local device configuration, and an operator can define the types of messages to forward. Most systems also save Syslog-generated messages locally (at least temporarily); however, as discussed in [Chapter 1](#), having to verify each system individually, one by one, is not ideal.

Syslog is used in a variety of systems, from Cisco routers to Linux systems. In this section, we focus on the networking version. As shown in [Example 2-17](#), a Syslog message has the following syntax:

Timestamp %facility-severity-MNEMONIC: description

The parts of this syntax are as follows:

- *Timestamp* is the time when the Syslog message was generated.
- *facility* is a code that refers to the process that issued the message (see [Table 2-1](#)), and it is not normative. Cisco, for example, uses different codes (refer to [Example 2-16](#)).
- *severity* indicates the urgency level of the event (see [Table 2-2](#)) and ranges from 0 (most urgent) to 7 (least urgent).
- *MNEMONIC* is text that uniquely describes the message.
- *description* is a more detailed description of the event.

Example 2-17 Cisco Router Syslog Messages

```
12:06:46: %LINK-3-UPDOWN: Interface Port-channel1, changed state to up
12:06:47: %LINK-3-UPDOWN: Interface GigabitEthernet0/48, changed state to up
12:06:48: %LINEPROTO-5-UPDOWN: Line protocol on Interface Vlan100, changed state to up
12:06:48: %LINEPROTO-5-UPDOWN: Line protocol on Interface GigabitEthernet0/48, changed state to down
```

```
18:47:02: %SYS-5-CONFIG_I: Configured from console by vty2 (10.0.0.0)
```

```
< [REDACTED] >
```

Table 2-1 *RFC 5424 Common Syslog Facilities*

Code	Facility
0	Kernel messages
1	User-level messages
2	Mail system
3	System daemons
4	Security/authorization messages
5	Messages generated internally
6	Line printer subsystem
7	Network news subsystem
8	UCCP subsystem
9	Clock daemon
10	Security/authorization messages
11	FTP daemon
12	NTP daemon

Table 2-2 *Syslog Severities*

13	Log audit	
14	Log alert	
Severity	Code	Meaning
Emergencies	0	System is unusable
Alerts	1	Action must be taken immediately
Critical	2	Critical conditions
Errors	3	Error conditions
Warnings	4	Warning conditions
Notifications	5	Normal but significant condition
Informational	6	Informational messages
Debugging	7	Debug-level messages

Syslog is commonly transported using UDP port 514. Because UDP is an unreliable protocol, some Syslog messages may be lost.

Tip

Knowing the format of Syslog messages can be very helpful when it comes to parsing them and working with automation actions.

NetFlow

NetFlow, which is a protocol created by Cisco, allows you to collect and export information about traffic flows passing on network equipment. It mainly consists of three components:

- **Flow exporter:** The flow exporter runs on network equipment. It collects and aggregates flow data and sends it to the flow collector (using a push model).
- **Flow collector:** The flow collector is typically a dedicated central component, such as a server, running elsewhere, and it stores the received information.
- **Analysis application:** The analysis application is what you use to

communicate with the flow collector. You can poll specific information and act on the results.

[Figure 2-1](#) shows the architecture of NetFlow.

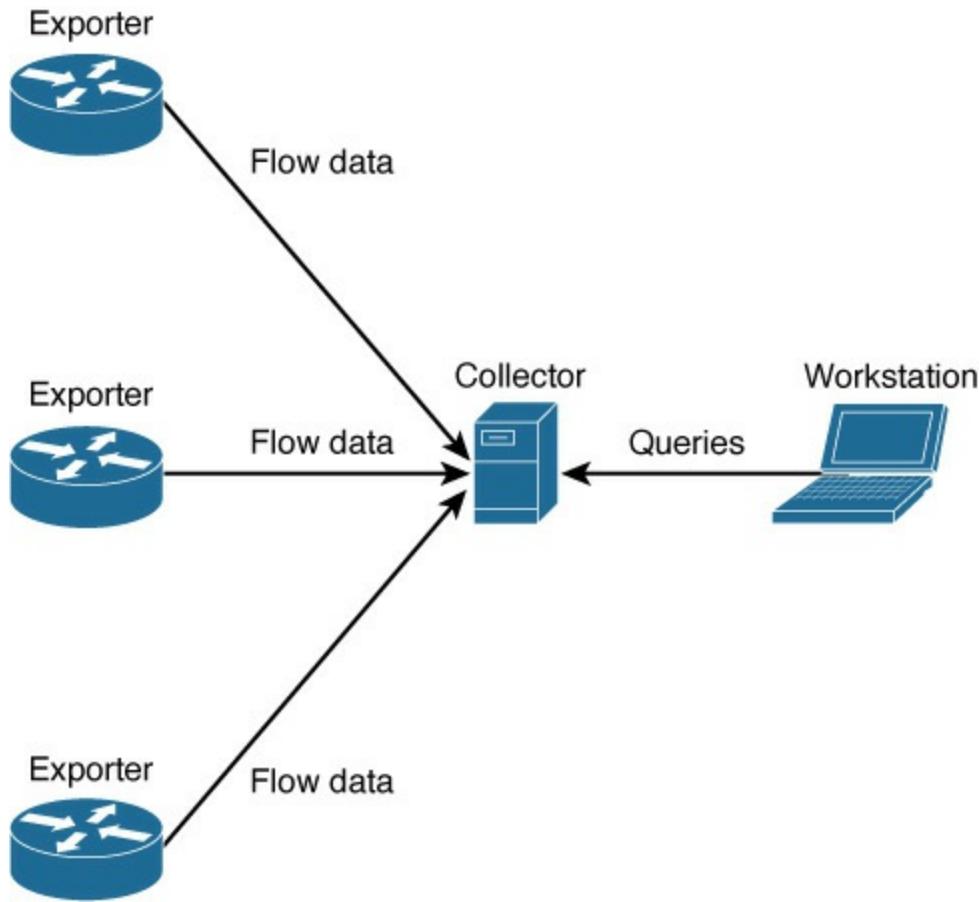


Figure 2-1 *NetFlow Log Data Common Architecture*

There are several versions of NetFlow. The most commonly used versions are 5 and 9.

Network flows are highly granular; for example, flow records include details such as IP addresses, packet and byte counts, timestamps, type of service (ToS), application ports, and input and output interfaces.

NetFlow has been superseded by Internet Protocol Flow Information Export (IPFIX), which is covered in the next section. However, NetFlow is still in use in many networks and is also valuable in network automation based on traffic flows.

Note

Refer to RFC 3954 for the NetFlow Version 9 packet and header format.

In summary, these are the important things to know about NetFlow:

- It is used for flow data collection.
- The data format may vary due to NetFlow's template-based nature.
- It is typically parsed by prebuilt collecting tools (flow collectors).

IPFIX

Internet Protocol Flow Information Export (IPFIX) was created by an IETF working group as an open standard for exporting flow information from network equipment. Just like NetFlow, it functions in a push model, where network devices push flow information to a central component, typically a server.

IPFIX architecture is the same as NetFlow's (refer to [Figure 2-1](#)).

A flow record contains information about a specific flow that was observed. It contains measured properties of the flow (for example, the total number of bytes for all of the flow's packets) and usually characteristic properties of the flow (such as the source IP address).

IPFIX is template based. The template defines the structure and semantics of the information that needs to be communicated between the devices and their respective collectors. This allows for granular control of the information gathered.

By using flow data, you can, for example, create alerts and baselines. Flow data also plays an important role in understanding what type of traffic is actually going through a network. You might think you know about the traffic, but analysis of flow data commonly brings surprises (such as misbehaving applications communicating on weird ports and incorrect firewall rules).

Note

Refer to RFC 5101 for the IPFIX packet and header format.

Cloud Flows

IPFIX and NetFlow give you visibility into traffic flows, but you need to implement these protocols on network devices that are on the traffic flow path. In the cloud, most of the time you do not use your own network devices but instead rely on the cloud provider to route your traffic.

To provide visibility into cloud traffic flows, cloud providers have flow logs. Using flow logs is better than having to install monitoring agents on all your endpoints.

You enable flow logs on a network basis, which means logging all traffic in a network where logging is enabled. There are options to ignore specific types of traffic based on characteristics such as protocol or IP address.

Like IPFIX, flow logs have a template format, which means you can make changes to suit your needs. The AWS default template has the following syntax:

```
$version $account-id $interface-id $srcaddr $dstaddr $srcport $dstpo  
$start $end $action $log-status
```



[Figure 2-2](#) shows several rows of a flow log. Each row represents a flow with the default AWS format.

	Timestamp	Message
		No older events at this moment. Retry
▶	2021-07-11T12:21:56.000+02:00	2 140412307939 eni-0c7a2fecdf7769741 89.248.165.122 10.0.128.135 44639 4002 6 1 40 1625998916 1625998972 REJECT OK
▶	2021-07-11T12:21:56.000+02:00	2 140412307939 eni-0c7a2fecdf7769741 10.0.141.246 10.0.128.135 9090 36436 6 5 2316 1625998916 1625998972 ACCEPT OK
▶	2021-07-11T12:21:56.000+02:00	2 140412307939 eni-0c7a2fecdf7769741 10.0.128.135 10.0.141.246 36436 9090 6 6 452 1625998916 1625998972 ACCEPT OK
▶	2021-07-11T12:21:56.000+02:00	2 140412307939 eni-0c7a2fecdf7769741 10.0.141.246 10.0.128.135 9090 36504 6 5 2316 1625998916 1625998972 ACCEPT OK
▶	2021-07-11T12:21:56.000+02:00	2 140412307939 eni-0c7a2fecdf7769741 10.0.128.135 10.0.141.246 36504 9090 6 5 400 1625998916 1625998972 ACCEPT OK
▶	2021-07-11T12:21:56.000+02:00	2 140412307939 eni-0c7a2fecdf7769741 10.0.141.246 10.0.128.135 9090 36472 6 5 2316 1625998916 1625998972 ACCEPT OK
▶	2021-07-11T12:21:56.000+02:00	2 140412307939 eni-0c7a2fecdf7769741 10.0.128.135 10.0.141.246 36472 9090 6 5 400 1625998916 1625998972 ACCEPT OK
▶	2021-07-11T12:21:56.000+02:00	2 140412307939 eni-0c7a2fecdf7769741 183.136.225.12 10.0.128.135 40724 9418 6 1 44 1625998916 1625998972 REJECT OK
▶	2021-07-11T12:21:56.000+02:00	2 140412307939 eni-0c7a2fecdf7769741 1.53.60.115 10.0.128.135 35324 23 6 1 40 1625998916 1625998972 REJECT OK
▶	2021-07-11T12:21:56.000+02:00	2 140412307939 eni-0c7a2fecdf7769741 162.142.125.157 10.0.128.135 5482 8118 6 1 44 1625998916 1625998972 REJECT OK
▶	2021-07-11T12:21:56.000+02:00	2 140412307939 eni-0c7a2fecdf7769741 89.248.165.122 10.0.128.135 44639 3458 6 1 40 1625998916 1625998972 REJECT OK

Figure 2-2 AWS Traffic Flow Logs

The central destination of flow logs is determined by the cloud providers, although most cloud providers offer the option of exporting these logs elsewhere, if required.

YANG

So far in this chapter, we have covered data formats and their respective data models. YANG is different: It is a data modeling language that is independent of the data format. When this book talks about YANG, it always mentions it in the context of network configuration or operational data. YANG provides a well understood and agreed upon structured data representation that is automation friendly. Different platforms from different vendors may implement the same YANG models, enabling standardized access to information.

JSON and XML schemas help limit data types and define and document available data. YANG serves the same purpose. In addition, it is an open

standard, it can be encoded in JSON or XML, it is very much focused on networking, and it is human readable.

For example, a data model can enforce IP addresses in the format [1–255].[1–255].[1–255].[1–255] or VLANs in the range 1 to 4094. It can also document in a generic format what data is available for retrieval, helping you understand what data you can get from a device. YANG does this better than other data modeling tools by having predefined data types such as IPv4 and MAC addresses.

YANG is very important for networking automation. Besides the benefits already mentioned, YANG is currently the model used in network devices to define the valid structure for data exchanged using NETCONF and RESTCONF, which are the “next-generation CLI” for device interaction (as discussed later in this chapter).

Let’s look at YANG components. The top-level construct in YANG is the module. A module defines a data model in hierarchical format. As shown in [Example 2-18](#), a module typically refers to a single feature or operational state.

Example 2-18 Simplified YANG Module (*ietf-interface*)

```
module ietf-interfaces {

    container interfaces {
        list interface {
            key "name";
            leaf name {
                type string;
            }
            leaf description {
                type string;
            }
        }
    }

    container interfaces-state {
        list interface {
```

```

leaf name {
    type string;
}
leaf enabled {
    type boolean;
    default "true";
}
}
}

```

YANG modules can be authored by standards bodies (for example, IETF, IANA) or by vendors (for example, Cisco, Juniper). The code in [Example 2-18](#) is a simplified version of an IETF model for a network interface.

Each YANG-capable device has support for the modules the vendor has decided to implement. Typically, network devices support the standard bodies' modules as well as the own vendors' modules.

Inside a module, there are four possible node types:

- **Container node:** In [Example 2-18](#), you can see two containers: *interfaces* and *interfaces-state*. A container groups related nodes within a subtree. It can contain any number of child nodes, independent of type.
- **Leaf node:** A node such as *name* or *description* is a leaf that contains a single data value.
- **List node:** In [Example 2-18](#), you can see two lists called *interface*. A list is used to define a sequence of entries. Each entry is uniquely identified by key. In this example, each interface is uniquely identifiable by its name.
- **Leaf list node:** Leaf lists are different from lists. A leaf list is a list of leafs, each with exactly one value instead of the possibility of multiple entries.

[Example 2-19](#) shows a YANG container with leaf-list.

Example 2-19 YANG Leaf List

```
container dhcp-server {
    container options {
        leaf-list dns-servers {
            type inet:ipv4-address;
            max-elements 8;
        }
    }
}
```

The node types will become clearer when you see data encoded in XML or JSON for YANG models.

Note

Typically, you do not have to create your own YANG models. However, you should understand their syntax in order to be able to interact with devices using NETCONF or RESTCONF.

As mentioned earlier, YANG is encoded using XML or JSON. Examples 2-20 and 2-21 show data encoded for the YANG model shown in Example 2-18.

Example 2-20 Sample XML Representation of the Module ietf-interface

```
<interfaces>
<interface>
    <name>Ethernet1/1</name>
    <description>test</description>
</interface>
<interface>
    <name>Ethernet1/2</name>
    <description>production</description>
</interface>
<interface>
    <name>Ethernet1/3</name>
    <description>staging</description>
```

```

        </interface>
    </interfaces>
<interfaces-state>
    <interface>
        <name>Ethernet1/1</name>
        <enabled>False</enabled>
    </interface>
    <interface>
        <name>Ethernet1/2</name>
        <enabled>True</enable>
    </interface>
    <interface>
        <name>Ethernet1/3</name>
        <enabled>False</enable>
    </interface>
</interfaces-state>

```

Example 2-21 Sample JSON Representation of the Module *ietf-interface*

```
{
    "interfaces": {
        "interface": [
            {
                "name": "Ethernet1/1",
                "description": "test"
            },
            {
                "name": "Ethernet1/2",
                "description": "production"
            },
            {
                "name": "Ethernet1/3",
                "description": "staging"
            }
        ]
    },
    "interfaces-state": {

```

```

    "interface": [
        {
            "name": "Ethernet1/1",
            "enabled": false
        },
        {
            "name": "Ethernet1/2",
            "enabled": true
        },
        {
            "name": "Ethernet1/3",
            "enabled": false
        }
    ]
}
}

```

In [Example 2-21](#), you can see that leaf statements are represented as key/value pairs. A container simply maps to an element in the hierarchy.

A leaf list is represented as a single element. The leaf list representation of [Example 2-19](#) in XML and JSON is as follows:

XML:

```

<dhcp-server>
    <options>
        <dns-servers>10.0.0.1</dns-servers>
        <dns-servers>10.0.0.2</dns-servers>
    </options>
</dhcp-server>

```

JSON:

```
{
    dhcp-server: {
        options: {
            dns-servers: [
                "10.0.0.1",
                "10.0.0.2"
            ]
        }
    }
}
```

```
        ]  
    }  
}  
}
```

You will see more examples of these encodings in NETCONF and RESTCONF examples, later in this chapter.

Methods for Gathering Data

Gathering data is an important step in making informed decisions. As you have already seen, using the right tool for the job is also an important step. Networks have different types of data with distinct formats, including the following:

- Device configurations
- Device metrics
- Flow data
- Log data

Using APIs is a common way of gathering device configurations. Model-driven telemetry is great for metrics, especially when you must act quickly on deviations. Log data, on the other hand, is great for anticipating impacting events, and it is commonly exported to central places using log exporters and further analyzed there. SSH still has a place when other methods are not available or for quick-and-dirty action, but it is slowly being replaced by other methods.

Choosing the most suitable method can be difficult, and the choice depends on use case. The following sections help you know how to choose the right method.

Note

The following sections do not cover SSH or SNMP, as they are well known at the time of this writing.

APIs

An application programming interface (API) represents a fixed set of instructions for how to interact with an application or a tool. The most common type of API is a REST (Representational State Transfer) API, but other types exist as well, such as SOAP (Simple Object Access Protocol) APIs. This chapter focuses on REST APIs as they are the most popular and the ones you will most likely come into contact with.

REST APIs are based in URIs (uniform resource identifiers, of which a URL is a specific type). The REST framework uses HTTP and is text based and stateless. *Stateless* means every request must contain all the information needed to perform the request, and it may not take advantage of information previously stored in the server. [Figure 2-3](#) shows an example of a REST API interaction.

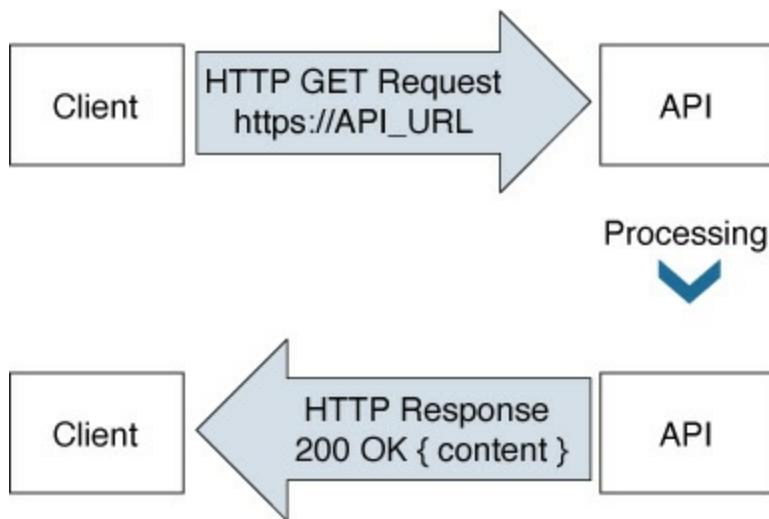


Figure 2-3 REST API GET Data Example

REST APIs use HTTP methods for the types of operations that are possible (see [Table 2-3](#)).

Table 2-3 HTTP Methods

HTTP Verb	Purpose	Description
GET	Read	Retrieve resource details from the system.
POST	Create	Create a new resource.
PUT	Update	Replace a resource. Can also create resources.
PATCH	Update	Modify partial details of a resource.
DELETE	Delete	Remove a resource from a system.

The biggest confusion is typically about the difference between PATCH and PUT. PATCH is used for partial updates, and PUT is used for overwriting an entire entity (see [Example 2-22](#)).

Example 2-22 HTTP PATCH Versus PUT Operations

```

Initial Data
{
  "name": "SWMAD01",
  "IP": "10.10.10.1",
  "status": "production",
}

PATCH Payload
{
  "IP": "10.10.10.2",
}

Resulting Data
{
  "name": "SWMAD01",
  "IP": "10.10.10.2",
  "status": "production",
}

Initial Data
{
  "name": "SWMAD01",
  "IP": "10.10.10.1",
  "status": "production",
}

PUT Payload
{

```

```

    "IP": "10.10.10.2",
}
Resulting Data
{
    "IP": "10.10.10.2",
}

```

[Example 2-22](#) shows JSON being used as a data format. It is the most common data format in REST APIs, but XML and YAML are also possible formats.

When interacting with an API, it is important to understand the status codes that are returned. The status code indicates what actually happened with a petition. [Table 2-4](#) lists the most common HTTP status codes returned when interacting with APIs.

Table 2-4 Common HTTP Status Codes

Status Code	Status Message	Description
200	OK	All good
201	Created	New resource created
202	Accepted	Accepted, but processing is not completed
204	No Content	Request is good, but no message body was returned
400	Bad Request	Request is invalid
401	Unauthorized	Authentication is missing or incorrect
403	Forbidden	Request was okay, but it is not allowed
404	Not Found	Resource is not found
500	Internal Server Error	Something is wrong with the server
503	Service Unavailable	Server is unable to complete the request

How do you actually interact with APIs? There are multiple ways, including the following:

- Postman
- **curl**

- Programming and scripting languages (for example, Python, Bash)
- Ansible

[Example 2-23](#) provides an example of using the command-line utility **curl**. This example shows a login token being retrieved from a Cisco DNAC sandbox.

Example 2-23 Retrieving a DNAC Token by Using curl

```
$ curl -k -u devnetuser:Cisco123! -X POST https://sandboxdnac.cisc
{"Token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiI1ZTlkYmI
GUiLCJhdXRoU291cmNlIjoiaW50ZXJuYWwiLCJ0ZW5hbnROYW1lIjoive5UMCIsInJ
TQ4NWM1MDA0YzBmYjIxMiJdLCJ0ZW5hbnRJZCI6IjVkyzQ0NGQzMTO4NWM1MDA0YzB
DY0NjcwNSwiaWF0IjoxNjA4NjQzMta1LCJqdGkiOiI0ZWQ2NzEyNy1jMDc4LTRkYTg
jciLCJ1c2VybmFtZSI6ImRldm5ldHVzZXIifQ.MiYK111psXvPnYf-
wISCNrx4pkYqZsdQppwXuZ69dL2e6342Ug9ahGdN8Vge12ww24YaUTY3BmWybMP0W7
gykDFHYM1c0bY6UbF6VhsayIP7P7bim07td8rrgLG0dv8PhUmYme-
jF9SuS1IIHyXPU1vfm3OQV8rIh_ZXXZtMp3W7MMfnBWlyQTxIeFBe3HpbEJZcxzfzb
jm6ELq4CZlnR_viidVgbqFUP2W49ks_o93Xhf5O3_XYaSXN7-
NIVsfadowYyec8R8JCKW4sMlqc0QIkWwhTr8zRfx1IN3TTQ6Q"}
```

In the **curl** command, you indicate the username (**devnetuser** in this example) and the password (**Cisco123!**), along with the HTTP method (**POST**), and the URI. You get a JSON response with a token.

[Figure 2-4](#) shows how you can retrieve this same token by using Postman.

The screenshot shows the Postman interface with the following details:

- Method:** POST
- URL:** https://sandboxdnac.cisco.com/dna/system/api/v1/auth/token
- Authorization:** Basic Auth (Username: devnetuser, Password: Cisco123!)
- Response Status:** 200 OK
- Response Body (Pretty JSON):**

```

1 {
2   "Token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.
  eyJzdWIiOiIzTlkYmI3NzdjZDQ3ZTAwNGM2N2RkMGUiLCJhdXR0U291cmNljoiaW50ZXJuYWwiLCJ0ZW5hbnROYW1lIjoive5UMCisInJvbGVzIjpbiJvKyZQ0NGQ1MTQ4NWM1MDA0YzBmYjIxMiJdLCJ0ZW5hbnRJZCI6IjVKyZQ0NGQzMTQ4NWM1MDA0YzBmYjIwYiIsImV4cCI6MTYxMTIyOTIxMiwiaWF0IjoxNjExMjI1NjEyLCJqdGkiOiIjODA0NjhsNS1iYzk0LTQ2NDktYmJhZi02MzgwODJmZmJiNjUiLCJ1c2VybmtZSI6ImRldm5ldHVzZXIifQ.
  VXCTRxa14biZQo_pPYWlrsnH06hn5AAP1oBYKggckMLzsory6rZpPwqP56qyTzR36Y5K5040jAT4AJUhFq6SVUXKOUC7bwZ_8iP0WnzYdsMifCTcmcmMch3rXchidraH1NeHrh7n51f1AnCMGQhEoiiii53NPc2WnT36Gwn7hv1r1fuG N70R0R0R0nn118iXwirCv nnk1uhhn05AcYvtHunhQRv"

```

Figure 2-4 REST API GET Using Postman

You can see in the top left of this figure the HTTP method POST, followed by the URI. At the bottom right you see the response code 200, as well as the received token.

Note

You can try using **curl** from your machine by using the same credentials and URI shown in [Example 2-23](#). You will get a different returned response token.

APIs are defined per product and system. There is no need to memorize how they work, as they all work differently. They have different URI paths and different payloads, but they are normally documented.

Tip

For a network system that you own, such as DNA Center, try to find its API documentation and execute an API call by using your preferred tool.

Now you know what REST APIs are and how they function. How can you use them in the context of network automation data gathering? Using APIs is an easy way to expose data in a structured format that can be easily parsed. Unlike CLIs, APIs are meant for programmatic access. Many networking products today expose APIs that you can interact with in order to retrieve information or change configurations.

The following are some examples of products with API support:

- Cisco DNA Center
- Any public cloud provider's service
- Firepower Management Center
- Cisco NX-OS equipment

You might be thinking that you can also use SSH to visit these systems and retrieve the needed information. For example, if you want to retrieve the interface status and MAC address from a router, you can connect and issue the **show interfaces** CLI command, as shown in [Example 2-24](#). However, you must programmatically parse the highlighted line from the whole output (by using regex, for example).

Example 2-24 *Using the CLI to Get Interface Information*

```

Router# show interfaces
Ethernet 0 is up, line protocol is up
  Hardware is MCI Ethernet, address is 0000.0c00.750c (bia 0000.0c
    Internet address is 131.108.28.8, subnet mask is 255.255.255.0
    MTU 1500 bytes, BW 10000 Kbit, DLY 100000 usec, rely 255/255, lo
    Encapsulation ARPA, loopback not set, keepalive set (10 sec)
    ARP type: ARPA, ARP Timeout 4:00:00
    Last input 0:00:00, output 0:00:00, output hang never
    Last clearing of "show interface" counters 0:00:00
    Output queue 0/40, 0 drops; input queue 0/75, 0 drops
    Five minute input rate 0 bits/sec, 0 packets/sec
    Five minute output rate 2000 bits/sec, 4 packets/sec
      1127576 packets input, 447251251 bytes, 0 no buffer
      Received 354125 broadcasts, 0 runts, 0 giants
      0 input errors, 0 CRC, 0 frame, 0 overrun, 0 ignored, 0 abort
      5332142 packets output, 496316039 bytes, 0 underruns
      0 output errors, 432 collisions, 0 interface resets, 0 restart

```

Say that there is an API available, and you can make a request such as the one in [Example 2-25](#). Parsing the API's response is much simpler than parsing the CLI's response. However, there are some cases in which SSH is required because there is no API available on the system or for the information you need.

Example 2-25 Using an API to Get Interface Information

```

$ curl https://Router/api/v1/device/interface
{
  "response": {
    "ifName": "Ethernet0",
    "macAddress": "00:00:0c:00:75:0c",
    "status": "up",
    "adminStatus": "UP",
  }
}

```

APIs can also be used in machine-to-machine communications or even between components on the same machine. These are sometimes referred to as *API integrations*.

Note

You should use APIs instead of the CLI whenever they are available.

Model-Driven Techniques

We have previously touched on the YANG data model. Network devices expose YANG models composed of configuration and operational data. They are automation friendly compared to using the CLI because data is exposed in a structured format, and multiple types of equipment may use the same models, overcoming the difficulties of different syntaxes. [Figure 2-5](#) shows different vendor devices implementing the same YANG model, resulting in an easily programmable vendor-agnostic environment. Such an environment is not completely possible yet, but efforts are being made in this area. In this section you will see how to interact with these models by using NETCONF and RESTCONF.

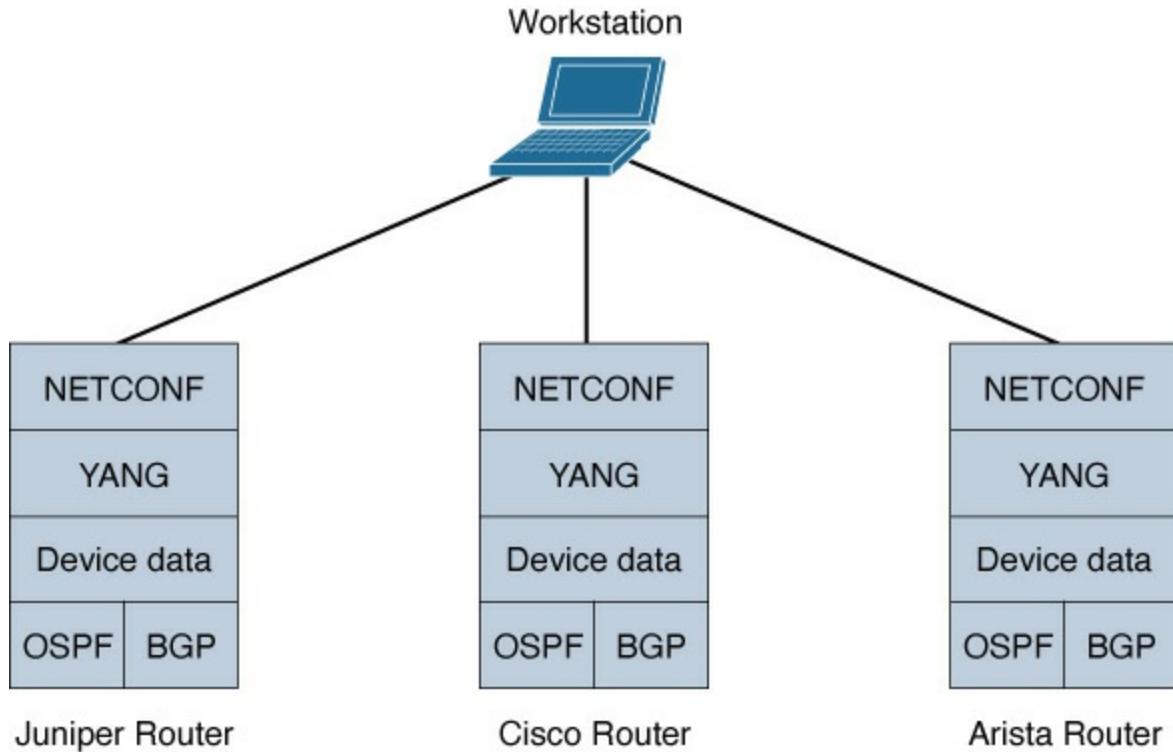


Figure 2-5 Different Platforms Exposing the Same YANG Model

NETCONF

NETCONF is a connection-oriented network management protocol developed by the IETF. It allows you to configure, delete, or retrieve data from network devices that support it. It uses XML for encoding and SSH for transport, using port 830. It works using a client/server architecture, where the server is a process running on the network devices you interact with.

There are two separate types of data when using NETCONF: configuration data and operational data. Configuration data is writable (for example, OSPF configuration or NTP). Operational data is read only (for example, interface or protocol statistics).

NETCONF uses the concept of multiple datastores. In networking, a device typically has a running configuration and a startup configuration. In NETCONF terms, those configurations are datastores. NETCONF also introduces a third datastore, called a *candidate datastore*. The same concept is used with Cisco IOS XR devices, where you enter all your configuration commands, but they do not take effect until you commit the changes.

[Table 2-5](#) summarizes NETCONF datastores and their uses.

Table 2-5 *NETCONF Datastores*

Datastore	Description
Running	Always exists. Represents the entire currently active configuration.
Candidate	Creates and changes configurations, with a commit operation this is applied to the running datastore. May or may not exist.
Startup	Represents the configuration used by a device when it boots up. May or may not exist.

Note

Not all devices support all three datastores. Some devices might even support only one.

As mentioned earlier, NETCONF is a network management protocol. To achieve its goals, it uses the series of operations described in [Table 2-6](#).

Table 2-6 *NETCONF Operations*

Operation	Description
get	Retrieves running configuration and device state information.
get-config	Retrieves all or part of a specified configuration datastore.
edit-config	Loads all or part of a specified configuration to the specified target configuration datastore. It supports several operation types: merge, replace, create, delete, and remove.
delete-config	Deletes a configuration datastore. The running configuration datastore cannot be deleted.
copy-config	Creates or replaces an entire datastore with the contents of another complete datastore.
lock/unlock	Locks or unlocks the configuration data store.
commit	Sets the running configuration to the current contents of the candidate configuration datastore.
validate	Validates the contents of the specified configuration.
close-session	Gracefully terminates a NETCONF session.
kill-session	Forces the termination of a NETCONF session
hello	Exchanges YANG capabilities

There are multiple ways to interact with NETCONF, including the following:

- ncclient Python library
- Ansible
- SSH

You will see Ansible examples in [Chapter 5](#), and I don't recommend using SSH to interact with NETCONF in a production environment because it is prone to errors; you can, however, use it for testing purposes.

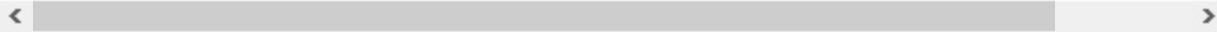
The flow of NETCONF communication goes like this:

- Step 1.** The client connects to the target device.
- Step 2.** The server sends a hello message about its capabilities to the client.
The client replies with a similar hello.
- Step 3.** The client sends operations, considering available capabilities.

Step 4. The server responds.

In step 2, the server advertises supported YANG modules along with native capabilities (such as support for the candidate datastore). This is key to enabling the client to know what it might be able to interact with. It has a format similar to the following, although it will vary depending on the device's capabilities:

```
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<capabilities>
<capability>urn:ietf:params:netconf:base:1.1</capability>
<capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
<capability>urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring</capability>
<capability>urn:ietf:params:xml:ns:yang:ietf-interfaces</capability>
$$$$</capabilities>
```



Steps 3 and 4 may be repeated several times in the same session.

The only piece of information you are missing regarding NETCONF is how to build and understand the XML payloads exchanged.

The payload starts and ends with an `<rpc>` tag because the whole process is based on remote procedure calls (RPCs). Messages are identified with the attribute *message-id* on this same tag. The second tag is the operation tag (for example `<get>` or `<get-config>`). The following XML snippet shows the payload to retrieve a device's running datastore:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
<get-config>
<source>
<running/>
</source>
</get-config>
</rpc>
```

Typically, when retrieving data from a device, the `<filter>` tag is used to limit the result to only relevant parts. For example, you could use the filter shown in [Example 2-26](#) to retrieve only the contents of the Gigabit Ethernet 1/0/13 interface.

Example 2-26 NETCONF Interface Filter

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1
  <get-config>
    <source>
      <running/>
    </source>
    <filter>
      <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfac
        <interface>
          <name>GigabitEthernet1/0/13</name>
        </interface>
      </interfaces>
    </filter>
  </get-config>
</rpc>
```

Executing this on a Cisco Catalyst 9300 running Version 16.12.04 would result in the snippet in [Example 2-27](#). Note in this example that the replies are enclosed in `<data>` tags.

Example 2-27 NETCONF RPC Reply with a Filter

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message
  <data>
    <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfac
      <interface>
        <name>GigabitEthernet1/0/13</name>
        <description>Fabric Physical Link</description>
        <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-
type">ianaift:ethernetCsmacd</type>
        <enabled>true</enabled>
        <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
          <address>
            <ip>172.31.63.165</ip>
            <netmask>255.255.255.254</netmask>
          </address>
        </ipv4>
      </interface>
    </interfaces>
  </data>
</rpc-reply>
```

```
        <ipv6 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip" />
    </interface>
</interfaces>
</data>
</rpc-reply>
```



You mostly need to be familiar with what is inside the operation tags, as the outer parts of the payload `<rpc>` and operation tags are typically abstracted by the tool you use to interact with a NETCONF-enabled device (unless you are using SSH directly). Say that you want to configure an interface description and an IP address on the interface Gigabit Ethernet 1/1 of an IOS XE device. You could achieve this by using the payload shown in [Example 2-28](#).

Example 2-28 NETCONF Code to Configure an Interface

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id=""
  <edit-config>
    <target>
      <running />
    </target>
    <config>
      <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-inter
        <interface>
          <name>GigabitEthernet1/1</name>
          <description>new description</description>
          <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
            <address>
              <ip>10.0.0.1</ip>
              <netmask>255.255.255.0</netmask>
            </address>
          </ipv4>
        </interface>
      </interfaces>
    </config>
  </edit-config>
</rpc>
```





When using the ncclient Python module, you pass the payload shown in [Example 2-29](#).

Example 2-29 Python Payload to Configure an Interface Using NETCONF

```
<config>
    <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-inter
        <interface>
            <name>GigabitEthernet1/1</name>
            <description>new description</description>
            <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
                <address>
                    <ip>10.0.0.1</ip>
                    <netmask>255.255.255.0</netmask>
                </address>
            </ipv4>
        </interface>
    </interfaces>
</config>
```

In [Example 2-29](#), you can see that all the outer tags are gone. You inform the tool about the desired operation and datastore in the command itself. *edit_config* is the operation, and the *target* parameter represents the target datastore:

```
edit_config(xml_payload, target="running")
```

As you can see in the previous examples, the resulting data is automation friendly, and it is easy to parse and consume. However, NETCONF requires quite a change from the ways people are accustomed to doing things, and it was not well adopted at first. A further evolution was RESTCONF.

RESTCONF

RESTCONF brought a REST API interface to NETCONF and YANG. It is

stateless, uses HTTP as the transport method, and uses the HTTP methods described in [Table 2-7](#). APIs, as previously mentioned, are widely adopted in the software world, and RESTCONF allows users who are familiar with APIs to use them to easily interact with network devices. The tools you use to interact with RESTCONF are the same as the API tools: **curl**, Postman, Python, and so on.

Table 2-7 NETCONF URI Elements

Datastore	Description
ADDRESS	Specifies the IP address of the RESTCONF-capable target agent.
ROOT	Specifies the main entry point for RESTCONF requests. To discover it you can use https://device_IP/.well-known/host-meta .
RESOURCE	Typically specifies the value /data or /operations, representing the type of resource being accessed.
YANGMODULE and CONTAINER	Specifies the base model container being used.
LEAF	Specifies an individual element of a YANG container.
OPTIONS	(Optional) Specifies the optional parameters, such as type of content that impacts return results.

Unlike NETCONF data, RESTCONF data can be encoded using XML or JSON. The URI is an important element of REST APIs, and a RESTCONF URI is similar. It has the following format:

```
https://ADDRESS/ROOT/RESOURCE/ [ YANGMODULE:] CONTAINER/ LEAF[ ?OPTIONS]
```

Let us revisit the NETCONF example of a Cisco IOS XE device. The URI path to retrieve its configuration using the native YANG model is as follows:

```
https://device_IP/restconf/data/Cisco-IOS-XE-native:native/
```

To retrieve only the interface Gigabit 1/1/13, as shown earlier in [Example 2-26](#) using NETCONF, you make a GET request to the URI in [Example 2-30](#). (Note, in this example, that %2F is the URL encoding of the symbol /, as you cannot pass this symbol natively in a URI.)

Example 2-30 Retrieving Interface Information by Using RESTCONF

```
curl -u 'cisco:cisco123' --request GET 'https://{{device_IP}}/restconf/interfaces/interfaces/interface=GigabitEthernet1%2F0%2F13'

<interface xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces"
xmlns:if="urn:ietf:params:xml:ns:yang:ietf-interfaces">
  <name>GigabitEthernet1/0/13</name>
  <description>Fabric Physical Link</description>
  <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-type">ianaift:ethernetCsmacd</type>
  <enabled>true</enabled>
  <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
    <address>
      <ip>172.31.63.165</ip>
      <netmask>255.255.255.254</netmask>
    </address>
  </ipv4>
  <ipv6 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
    </ipv6>
  </interface>
```



You may notice that the result is exactly the same result obtained previously using NETCONF, as both examples use the same device. The structure represented on the URI maps directly to the YANG module used. In this example, you are using the ietf-interfaces YANG module and the interfaces container within it. The leaf is the interface named Gigabit Ethernet 1/1/13. You can see the importance of understanding a little bit of YANG.

To make configuration changes, you can use PUT, POST, or PATCH, as discussed earlier in this chapter. To replace the current interface configuration with a different one, for example, we could execute the PUT operation by using **curl**, as shown in [Example 2-31](#). Notice that the changes go in the payload, and the URI is maintained.

Example 2-31 Modifying Interface Configurations by Using RESTCONF

```
curl --request PUT 'https://{{device_IP}}/restconf/data/ietf-
```

```

interfaces:interfaces/interface=GigabitEthernet1%2F0%2F13' \
--header 'Content-Type: application/yang-data+xml' \
-u 'cisco:cisco123' \
--data-raw '<interface xmlns="urn:ietf:params:xml:ns:yang:ietf-int
xmlns:if="urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <name>GigabitEthernet1/0/13</name>
    <description>Fabric Physical Link</description>
    <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-
type">ianaift:ethernetCsmacd</type>
    <enabled>true</enabled>
    <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
        <address>
            <ip>172.31.63.164</ip>
            <netmask>255.255.255.254</netmask>
        </address>
    </ipv4>
    <ipv6 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
    </ipv6>
</interface>'
```

< **>**

In this example, you receive an empty 204 response. Notice here that you have sent all the information for PUT, even the information you did not want to change; this is a replace operation, as explained earlier in this chapter, and so any omitted data will be replaced by its default value.

If you simply want to change the description of an interface, you can use PATCH and a simpler payload:

```

<interface xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces"
xmlns:if="urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <description>New description</description>
</interface>
```

By now you have a good understanding of NETCONF and RESTCONF. Figure 2-6 shows a summary of these two protocols, as well as the Google protocol Google RPC (gRPC), which you will see later in this chapter.

Protocol	NETCONF	RESTCONF	gRPC
Encoding	XML	JSON	GPB
Transport	SSH	HTTP	
Model	YANG		

Figure 2-6 Model-Driven Protocol Stack

Telemetry

This chapter has covered model-driven techniques from a data configuration standpoint. One area where these techniques are commonly used is with telemetry. Telemetry is not configuration data but metric data. In the past, SNMP was the de facto standard for retrieving this type of data, and CLI connection methods such as SSH were also used. However, some use cases, such as setting off alarms based on critical conditions, require instant action on data, and most methods cannot achieve such agility. Model-driven telemetry is a push-based mechanism (refer to [Chapter 1](#)) that addresses the shortcomings of polling mechanisms such as SNMP.

There are two categories of model-driven telemetry:

- **Dial-in:** Telemetry management systems subscribe to telemetry streams on the devices.
- **Dial-out:** A device configures the subscription.

In either category, telemetry-enabled devices constantly send telemetry data to their management systems, which store this data in database systems. This enables access to real-time, model-driven data, which can then be used in network automation systems.

There are both periodic and change-based notifications. That is, a management system can receive information constantly every specified amount of time, or it can be notified only when a change occurs in a specific model.

The most commonly used data model for network telemetry is YANG. Hence, the protocols used are NETCONF, RESTCONF, and gRPC.

Telemetry has not yet been widely adopted, but it is quickly taking off as vendors enable their devices for this type of access.

One way of configuring model-driven telemetry is directly on a device (dial-out). Consider the following example on a Cisco CSR 1000V:

```
csrv000v(config)#telemetry ietf subscription 102
csr1000v(config-mdt-subs)#encoding encode-kvvpb
csr1000v(config-mdt-subs)#filter xpath /process-cpu-ios-xe-oper:cpu-
csr1000v(config-mdt-subs)#stream yang-push
csr1000v(config-mdt-subs)#update-policy periodic 1000
csr1000v(config-mdt-subs)#receiver ip address 10.0.0.1 57500 protocol
< >
```

This example shows data being sent to a collector at 10.0.0.1, using gRPC and Google protocol buffers as encoding. Data on CPU utilization is being sent every 10 seconds (using the process-cpu-ios-xe-oper YANG module).

You can also configure telemetry subscriptions by using NETCONF or RESTCONF. At the time of this writing, only gRPC is supported for dial-out telemetry in IOS XE.

The collector must be configured to receive this type of data. Commonly used collectors are Telegraf and Splunk. After the data is received in the tool, it should be stored. Finally, from the stored data, you can create visualizations and alarms (for example, by using Grafana). You will see more about this topic in [Chapter 3, “Using Data from Your Network.”](#)

Tip

Verify whether your devices support model-driven telemetry. If they do, use model-driven telemetry instead of SNMP or the CLI.

Log Exporters

This section covers exporters for two types of data:

- Message logs
- Flow logs

Message logs are vast in networks. They can originate in networking equipment, servers, applications, and other places. Message data type is different from metrics in that it is more verbose and typically bigger in size. Independently of the format or type, logs are good indicators of system behaviors. Most systems are programmed to issue logs when events such as failures or alarming conditions occur.

In networking, the most common log format is Syslog, covered earlier in this chapter.

The way to set up log exports differs from system to system. However, most networking systems today support a Syslog destination server. A typical configuration on an end system, such as a Cisco router, would look as follows, with a destination server IP address:

```
Router(config) # logging host 10.0.0.1
```

The destination IP address should point at a central collection tool, such as one of the tools mentioned in [Chapter 1](#). The Syslog server can then process these logs and either display them all together in a dashboard or redirect the relevant ones to other tools; for example, it might send the security logs to the security team monitoring tool and the operational logs to a different tool.

If a network device, such as the SWITCH1234 interface Gigabit3/0/48, comes up and generates a Syslog message such as the one below, the receiving tool (in this case, Splunk) would display it like the first message in [Figure 2-7](#):

```
15:00:17: %LINEPROTO-5-UPDOWN: Line protocol on Interface GigabitEth  
 < [REDACTED] >
```



Figure 2-7 *Splunk Syslog View*

In your central receiving tool, you can use the logs to derive insights, or you can simply store them.

NetFlow and IPFIX produce flow log data. This type of log data is typically less verbose than other log data; however, some template used for IPFIX can have many fields and become as verbose as the typical log data. The setup of this type of system is similar to Syslog setup, where you define a log destination on the exporting device and traffic is Layer 3 encapsulated until it reaches the collector. At the collector, it is de-encapsulated, parsed, and stored. A typical configuration on an end system, such as a Cisco router, would look as follows:

```
flow exporter EXPORTER
destination 10.0.0.1
export-protocol ipfix
transport udp 90
```

```

!
flow monitor FLOW-MONITOR
record netflow ipv4 original-input
exporter EXPORTER

```

There are many possible receiving applications for this type of flow data. [Figure 2-7](#) shows a Cisco Tetration dashboard created based on flow data received from network devices. In it, you can see the provider and the consumer of the flow, the ports used, the protocol, and the type of addresses. It is possible to get even more information, depending on the export template used.

	Timestamp	Consumer Address	Provider Address	Consumer Port	Provider Port	Protocol	Address Type
	Jan 18 5:38:00pm	10.201.34.164	10.201.34.141	0	0	ARP_REQUEST	IPv4
	Jan 18 5:38:00pm	10.201.144.127	161.44.124.122	40932	53	UDP	IPv4
	Jan 18 5:38:00pm	10.201.147.119	10.201.147.100	27090	443	TCP	IPv4
	Jan 18 5:38:00pm	0.0.0.0	255.255.255.255	68	67	UDP	DHCPv4
	Jan 18 5:38:00pm	10.201.22.252	0.0.0.0	0	0	ARP_REQUEST	IPv4
	Jan 18 5:38:00pm	10.201.145.55	0.0.0.0	0	0	ARP_REQUEST	IPv4
	Jan 18 5:38:00pm	fe80::d6e8:80ff:fe9:8fa4	ff02::1:2	546	547	UDP	IPv6
	Jan 18 5:38:00pm	fe80::1c4a:a68c:de:74d3	ff02::1:3	59377	5355	UDP	IPv6
	Jan 18 5:38:00pm	10.201.16.130	10.201.16.129	0	0	ARP_REQUEST	IPv4
	Jan 18 5:38:00pm	224.0.0.2	10.201.34.126	1985	1985	UDP	IPv4

Figure 2-8 *Tetration Flow Dashboard*

Beats, which is mentioned briefly in [Chapter 1](#), is another log exporter. It is different from the other log exporters discussed here in that it is agent based,

which means you must install a specific software artifact; in contrast, Syslog and NetFlow/IPFIX are natively present on the equipment. Beats enables you to export logs from systems that do not have inherent Syslog or NetFlow/IPFIX logging capabilities and support installation of an agent. In networking you will often use Syslog instead of Beats.

The tool you should use depends on the type of logs you need to export and the system in question. Choosing a tool can be a hard task. One tip for making it easier is to start by considering what types of log data you want to export, which should automatically reduce the list of candidates significantly.

Networks are complex and entail many interlocking components. Because of this, logs are most effective when they are correlated. Logs from single components are typically less effective at finding the root cause of an issue. Having a centralized log location is key, and this is where automation tools like Kibana and Splunk can assist you (refer to [Chapter 1](#)).

Note

Exporting all logs and analyzing them in a workstation is *not* considered using a centralized location.

Furthermore, simply collecting and storing logs is not enough. Customers often overflow themselves and their systems with meaningless information. Consider exporting and storing only logs that you might use instead of exporting and storing debug level logs for every single component. Having a sea of logs will increase the time required for deduplication, correlation, and analysis and will make it more difficult to derive value from logs.

Summary

This chapter starts by showcasing the crucial role that data plays in our systems and its key role in network automation.

This chapter covers many of the important data formats and models used in networking today and looks at how those types are used in machine-to-machine and human-to-machine communications. You have seen that data

types do not enforce content types, and you must use data models for that. Understanding this is important for interacting with network devices.

This chapter provides an overview of APIs, including their functioning, uses, and benefits compared to CLI access and “next-generation” model-driven techniques.

This chapter also describes how to capture data such as metrics, logs, and configurations. In summary, you should keep in mind the following points:

- If you want to capture metrics and need instant action, model-driven telemetry (if available) is the best choice.
- If you simply want to capture metrics with more relaxed time to action, you can use SNMP, which is a more traditional option.
- For configurations, APIs or model-driven configuration methods (such as NETCONF and RESTCONF) tend to work best.
- To capture log data, using a log exporter for the log type is usually the correct choice.

The next chapter dives into how to use collected data to derive insights and business value.

End Notes

RFC 5424, *The Syslog Protocol*, R. Gerhards, Adiscon GmbH
IETF, <https://tools.ietf.org/html/rfc5424>, March 2019

RFC 3954, *Cisco Systems NetFlow Services Export Version 9*, B. Claise, Ed., Cisco Systems, Inc.,
<https://tools.ietf.org/html/rfc3954>, October 2004

RFC 5101, *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information*, B. Claise, Ed., Cisco Systems, Inc.,
<https://tools.ietf.org/html/rfc5101>, January 2008

Review Questions

You can find answers to these questions in [Appendix A, “Answers to Review Questions.”](#)

- 1.** Which of the following is not a YANG component?
 - a.** Container list
 - b.** List
 - c.** Leaf
 - d.** Container
- 2.** True or false: In YAML, file indentation is unimportant.
 - a.** True
 - b.** False
- 3.** Which of the following are YAML data types? (Choose two.)
 - a.** Scalar
 - b.** Dictionary
 - c.** Leaf
 - d.** Container
- 4.** Is the following YAML valid?

```
---  
devices_dictionary: {CSR1000v: {address: "10.0.0.1", os: "iosxe"},  
                    N7K: {address: "10.0.0.2", os: "nxos"} }
```

 - a.** Yes
 - b.** No
- 5.** True or false: All Netconf messages are enclosed within *<rpc>* tags.
 - a.** True

- b.** False
- 6.** If you are trying to update only the description of an interface by using RESTCONF, which HTTP method should you use?
- a.** PUT
 - b.** PATCH
 - c.** GET
 - d.** POST
- 7.** What encodings can RESTCONF use? (Choose two.)
- a.** GPB
 - b.** XML
 - c.** YAML
 - d.** JSON
- 8.** True or false: Syslog has a single format across all implementations.
- a.** True
 - b.** False
- 9.** You are tasked with automating the collection of logs from all your network devices. Which of these techniques is the best option suited?
- a.** SSH
 - b.** NETCONF
 - c.** Log exporter
 - d.** Python
- 10.** Which of the following are valid tools for interacting with APIs? (Choose two.)
- a.** curl

b. Postman

c. pyang

d. Splunk

Chapter 3. Using Data from Your Network

In [Chapter 2](#), “Data for Network Automation,” you learned about data types and data models, as well as common methods to gather data from your network infrastructure. After you have stored your network data, what do you do with it? Are you currently storing any of your network’s data (logs, metrics, configurations)? If so, which data? What for?

You typically store data in order to find insights in it and to comply with regulatory requirements. An insight, in this sense, can be any useful information or action.

This chapter helps you understand how to use the collected data from your network and derive value from it. In the context of enterprise networks, this chapter covers the following topics:

- Data preparation techniques
- Data visualization techniques
- Network insights

At the end of this chapter are some real case studies that are meant to inspire you to implement automation solutions in your own network.

Data Preparation

Data comes in various formats, such as XML, JSON, and flow logs (refer to [Chapter 2](#)). It would be nice if we could gather bread from a field, but we must gather wheat and process it into bread. Similarly, after we gather data from a variety of devices, we need to prepare it. When you have a heterogeneous network, even if you are gathering the same type of data from different places, it may come in different formats (for example, NetFlow on a Cisco device and IPFIX on an HPE device). Data preparation involves

tailoring gathered data to your needs.

There are a number of data preparation methods. Data preparation can involve simple actions such as normalizing the date format to a common one as well as more complex actions such as aggregating different data points. The following sections discuss some popular data preparation methods that you should be familiar with.

Parsing

Most of the times when you gather data, it does not come exactly as you need it. It may be in different units, it may be too verbose, or you might want to split it in order to store different components separately in a database. In such cases, you can use parsing techniques.

There are many ways you can parse data, including the following:

- **Type formatting:** You might want to change the type, such as changing seconds to minutes.
- **Splitting into parts:** You might want to divide a bigger piece of information into smaller pieces, such as changing a sentence into words.
- **Tokenizing:** You might want to transform a data field into something less sensitive, such as when you store payment information.

You can parse data before storage or when consuming it from storage. There is really no preferred way, and the best method depends on the architecture and storage capabilities. For example, if you store raw data, it is possible that afterward, you might parse it in different ways for different uses. If you store raw data, you have many options for how to work with that data later; however, it will occupy more storage space, and those different uses may never occur. If you choose to parse data and then store it, you limit what you store, which saves space. However, you might discard or filter a field that you may later need for a new use case.

Regular expressions (regex) play a big part in parsing. Regex, which are used to find patterns in text, exist in most automation tools and programming languages (for example, Python, Java). It is important to note that, regardless of the tool or programing language used, the regex are the same. Regex can

match specific characters, wildcards, and sequences. They are not predefined; you write your own to suit your need, as long as you use the appropriate regex syntax. [Table 3-1](#) describes the regex special characters.

Table 3-1 *Regex Special Characters*

Character	Meaning
\d	Matches characters that contain digits from 0-9.
\D	Matches characters that do not contain digits from 0-9.
\w	Matches any word containing characters from a to z, A to Z, 0 to 9, or the underscore character _
\W	Matches any non-word characters (not containing characters from a to z, A to Z, 0 to 9, or the underscore character _).
\s	Matches any white space character (spaces, tabs, newlines, carriage returns).
\S	Matches any non-white space character.

[Table 3-2](#) describes a number of regex meta characters; although this is not a complete list, these are the most commonly used meta characters.

Table 3-2 *Regex Meta Characters*

Characters	Meaning
[]	A set of characters
.	Any character
^	Starts with
\$	Ends with
+	One or more occurrences
*	Zero or more occurrences
{}	Exact number of occurrences
	OR

[Example 3-1](#) shows how to use regex to find the IP address of an interface.

Example 3-1 *Using Regex to Identify an IPv4 Address*

```
Given the configuration:  
interface Loopback0  
description underlay  
ip address 10.0.0.1 255.255.255.0
```

```
Regex expression:  
(?:[0-9]{1,3}\.){3}[0-9]{1,3}
```

```
Result:  
["10.0.0.1", "255.255.255.0"]
```

[Example 3-1](#) shows a simplex regex that matches on three blocks of characters that range from 0 to 999 and have a trailing dot, followed by a final block of characters ranging from 0 to 999. The result in this configuration is two entries that correspond to the IP address and the mask of the Loopback0 interface.

IP addresses are octets, with each unit consisting of 8 bits ranging from 0 to 255. As an exercise, improve the regex in [Example 3-1](#) to match only the specific 0 to 255 range in each IP octet. To try it out, find one of the many websites that let you insert text and a regex and then show you the resulting match.

You can also use a number of tools, such as Python and Ansible, to parse data. [Example 3-2](#) shows how to use Ansible for parsing. Firstly, it lists all the network interfaces available in a Mac Book laptop, using the code in the file all_interfaces.yml. Next, it uses the regex ^en to display only interfaces prefixed with en. This is the code in en_only.yml.

Example 3-2 Using Ansible to Parse Device Facts

```
$ cat all_interfaces.yml  
---  
- hosts: all  
  tasks:  
    - name: print interfaces  
      debug:  
        msg: "{{ ansible_interfaces }}"
```

```
$ ansible-playbook -c local -i 'localhost,' all_interfaces.yml
PLAY [all] ****
TASK [Gathering Facts]
*****
ok: [localhost]

TASK [print interfaces]
*****
ok: [localhost] => {
    "msg": [
        "awdl0",
        "bridge0",
        "en0",
        "en1",
        "en2",
        "en3",
        "en4",
        "en5",
        "gif0",
        "llw0",
        "lo0",
        "p2p0",
        "stf0",
        "utun0",
        "utun1"
    ]
}
PLAY RECAP ****
localhost                               : ok=2      changed=0      unreachable=0
rescued=0      ignored=0

$ cat en_only.yml
---
- hosts: all
  tasks:
    - name: print interfaces
      debug:
        msg: "{{ ansible_interfaces | select('match', '^en') | lis
```

```

$ ansible-playbook -c local -i 'localhost,' en_only.yml

PLAY [all] ****

TASK [Gathering Facts]
*****
ok: [localhost]

TASK [print interfaces]
*****
ok: [localhost] => {
    "msg": [
        "en0",
        "en1",
        "en2",
        "en3",
        "en4",
        "en5"
    ]
}

PLAY RECAP ****
localhost                  : ok=2      changed=0      unreachable=0
rescued=0      ignored=0
< >

```

What if you need to apply modifications to interface names, such as replacing *en* with *Ethernet*? In such a case, you can apply mapping functions or regex, as shown with Ansible in [Example 3-3](#).

Example 3-3 Using Ansible to Parse and Alter Interface Names

```

$ cat replace_ints.yml
---
- hosts: all
  tasks:
    - name: print interfaces

```

```
debug:
  msg: "{{ ansible_interfaces | regex_replace('en', 'Etherne"

$ ansible-playbook -c local -i 'localhost,' replace_ints.yml

PLAY [all] ****
TASK [Gathering Facts]
*****
ok: [localhost]

TASK [print interfaces]
*****
ok: [localhost] => {
    "msg": [
        "awdl0",
        "bridge0",
        "Ethernet0",
        "Ethernet1",
        "Ethernet2",
        "Ethernet3",
        "Ethernet4",
        "Ethernet5",
        "gif0",
        "llw0",
        "lo0",
        "p2p0",
        "stf0",
        "utun0",
        "utun1"
    ]
}
PLAY RECAP ****
localhost                  : ok=2      changed=0      unreachable=0
rescued=0     ignored=0
< >
```

Note

Ansible is covered in [Chapters 4, “Ansible Basics,”](#) and [5, “Using Ansible for Network Automation.”](#) When you finish reading those chapters, circle back to this example for a better understanding.

Another technique that can be considered parsing is enhancing data with other fields. Although this is typically done before storage and not before usage, consider that sometimes the data you gather might not have all the information you need to derive insights. For example, flow data might have SRC IP, DST IP, SRC port, and DST port information but no date. If you store that data as is, you might be able to get insights from it on the events but not when they happened. Something you could consider doing in this scenario is appending or prepending the current date to each flow and then storing the flow data.

As in the previous example, there are many use cases where adding extra data fields can be helpful—such as with sensor data that does include have sensor location or system logs that do not have a field indicating when maintenance windows take place. Adding extra data fields is a commonly used technique when you know you will need something more than just the available exported data.

[Example 3-4](#) enhances the previous Ansible code (refer to [Example 3-3](#)) by listing the available interfaces along with the time the data was collected.

Example 3-4 Using Ansible to List Interfaces and Record Specific Times

```
$ cat fieldadd.yml
---
- hosts: all
  tasks:
    - name: print interfaces
      debug:
        msg: "{{ ansible_date_time.date + ansible_interfaces | reg
'Ethernet' }}"
$ ansible-playbook -c local -i 'localhost,' fieldadd.yml
```

```

PLAY [all] ****
TASK [Gathering Facts]
*****
ok: [localhost]

TASK [print interfaces]
*****
ok: [localhost] => {
    "msg": "2021-01-22['awd10', 'bridge0', 'Ethernet0', 'Ethernet1',
'Ethernet3', 'Ethernet4', 'Ethernet5', 'gif0', 'llw0', 'lo0', 'p2p',
'utun1'] "
}

PLAY RECAP ****
localhost : ok=2     changed=0     unreachable=0
rescued=0   ignored=0
< >

```

As part of parsing, you might choose to ignore some data. That is, you might simply drop it instead of storing or using it. Why would you do this? Well, you might know that some of the events that are taking place taint your data. For example, say that during a maintenance window you must physically replace a switch. If you have two redundant switches in your architecture, while you are replacing one of them, all your traffic is going through the other one. The data collected will reflect this, but it is not a normal scenario, and you know why it is happening. In such scenarios, ignoring data points can be useful, especially to prevent outliers on later analysis.

So far, we have mostly looked at examples of using Ansible to parse data. However, as mentioned earlier, you can use a variety of tools for parsing.

Something to keep in mind is that the difficulty of parsing data is tightly coupled with its format. Regex are typically used for text parsing, and text is the most challenging type of data to parse. [Chapter 2](#) mentions that XQuery and XPath can help you navigate XML documents. This should give you the idea that different techniques can be used with different types of data.

[Chapter 2](#)'s message regarding replacing the obsolete CLI access with NETCONF, RESTCONF, and APIs will become clearer when you need to

parse gathered data. [Examples 3-5](#) and [3-6](#) show how you can parse the same information gathered in different formats from the same device.

Example 3-5 Using Ansible with RESTCONF to Retrieve an Interface Description

```
$ cat interface_description.yml
---
- name: Using RESTCONF to retrieve interface description
  hosts: all
  connection: httpapi

  vars:
    ansible_connection: httpapi
    ansible_httpapi_port: 443
    ansible_httpapi_use_ssl: yes
    ansible_network_os: restconf
    ansible_user: cisco
    ansible_httpapi_password: cisco123

  tasks:
    - name: Retrieve interface configuration
      restconf_get:
        content: config
        output: json
        path: /data/ietf-interfaces:interfaces/interface=GigabitEthernet1
      register: cat9k_rest_config

    - name: Print all interface configuration
      debug: var=cat9k_rest_config

    - name: Print interface description
      debug: var=cat9k_rest_config['response']['ietf-interfaces:interface'][0]['description']

$ ansible-playbook -i '10.201.23.176,' interface_description.yml
PLAY [Using RESTCONF to retrieve interface description]
```

```
*****
TASK [Retrieve interface configuration]
*****
ok: [10.201.23.176]

TASK [Print all interface configuration] ****
ok: [10.201.23.176] => {
    "cat9k_rest_config": {
        "ansible_facts": {
            "discovered_interpreter_python": "/usr/bin/python"
        },
        "changed": false,
        "failed": false,
        "response": {
            "ietf-interfaces:interface": [
                {
                    "description": "New description",
                    "enabled": true,
                    "ietf-ip:ipv4": [
                        {
                            "address": [
                                {
                                    "ip": "172.31.63.164",
                                    "netmask": "255.255.255.254"
                                }
                            ]
                        },
                        {
                            "ietf-ip:ipv6": {},
                            "name": "GigabitEthernet1/0/13",
                            "type": "iana-if-type:ethernetCsmacd"
                        }
                    ],
                    "warnings": []
                }
            }
        }
    }
}

TASK [Print interface description] ****
ok: [10.201.23.176] => {
    "cat9k_rest_config['response']['ietf-interfaces:interface'][0]['description']"
}
```

```
PLAY RECAP ****
10.201.23.176 : ok=3     changed=0     unreachable=0
rescued=0      ignored=0
```

In [Example 3-5](#), you can see that when using a RESTCONF module, you receive the interface information in JSON format. Using Ansible, you can navigate through the JSON syntax by using the square bracket syntax. It is quite simple to access the interface description, and if you needed to access some other field, such as the IP address field, you would need to make only a minimal change:

```
debug: var=cat9k_rest_config['response']['ietf-interfaces:interface']
```

[Example 3-6](#) achieves the same outcome by using a CLI module and regex.

Example 3-6 Using Ansible with SSH and Regex to Retrieve an Interface Description

```
$ cat interface_description.yml
---
- name: Retrieve interface description
  hosts: all
  gather_facts: no

  vars:
    ansible_user: cisco
    ansible_password: cisco123
    interface_description_regexp: "description [\\w+ *]*"

  tasks:
    - name: run show run on device
      cisco.ios.ios_command:
        commands: show run interface GigabitEthernet1/0/13
      register: cat9k_cli_config

    - name: Print all interface configuration
```

```

debug: var=cat9k_cli_config

- name: Scan interface configuration for description
  set_fact:
    interface_description: "{{ cat9k_cli_config.stdout[0] | regex.findall(interface_description_regex, multiline=True) }}"

- name: Print interface description
  debug: var=interface_description

$ ansible-playbook -i '10.201.23.176,' interface_description.yml
PLAY [Retrieve interface description]
*****

```

TASK [run show run on device]

```

ok: [10.201.23.176]

TASK [Print all interface configuration]
*****

```

```

ok: [10.201.23.176] => {
  "cat9k_cli_config": {
    "ansible_facts": {
      "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "deprecations": [
      {}
    ],
    "failed": false,
    "stdout": [
      "Building configuration...\n\nCurrent configuration : bytes\n!\ninterface GigabitEthernet1/0/13\n  description New descri
switchport\n  dampening\n  ip address 172.31.63.164 255.255.255.254\n  pim sparse-mode\n  ip router isis\n  load-interval 30\n  bfd interval
  multiplier 3\n  clns mtu 1400\n  isis network point-to-point\nend"
    ],
    "stdout_lines": [
      [

```

```

        "Building configuration...",
        """",
        "Current configuration : 347 bytes",
        "!",
        "interface GigabitEthernet1/0/13",
        " description New description",
        " no switchport",
        " dampening",
        " ip address 172.31.63.165 255.255.255.254 seconda
        " ip address 172.31.63.164 255.255.255.254",
        " no ip redirects",
        " ip pim sparse-mode",
        " ip router isis ",
        " load-interval 30",
        " bfd interval 500 min_rx 500 multiplier 3",
        " clns mtu 1400",
        " isis network point-to-point ",
        "end"
    ]
],
"warnings": []
}
}

```

```

TASK [Scan interface configuration for description]
*****
ok: [10.201.23.176]

```

```

TASK [Print interface description]
*****
ok: [10.201.23.176] => {
    "interface_description": [
        "description New description"
    ]
}

```

```

PLAY RECAP *****
10.201.23.176 : ok=4      changed=0      unreachable=0
rescued=0      ignored=0

```



[Example 3-6](#) uses the following regex:

```
"description [\\w+ ]*"
```

This is a simple regex example, but things start to get complicated when you need to parse several values or complex values. Modifications to the expected values might require building new regex, which can be troublesome.

By now you should be seeing the value of the structured information you saw in [Chapter 2](#).

Aggregation

Data can be aggregated—that is, used in a summarized format—from multiple sources or from a single source. There are multiple reasons you might need to aggregate data, such as when you do not have enough computing power or networking bandwidth to use all the data points or when single data points without the bigger context can lead to incorrect insights.

Let's look at a networking example focused on the CPU utilization percentage in a router. If you are polling the device for this percentage every second, it is possible that for some reason (such as a traffic burst punted to CPU), it could be at 100%, but then, in the next second, it drops to around 20% and stays there. In this case, if you have an automated system to act on the monitored metric, you will execute a preventive measure that is not needed. [Figure 3-1](#) and [3-2](#) show exactly this, where a defined threshold for 80% CPU utilization would trigger if you were measuring each data point separately but wouldn't if you aggregated the data and used the average of the three data points.

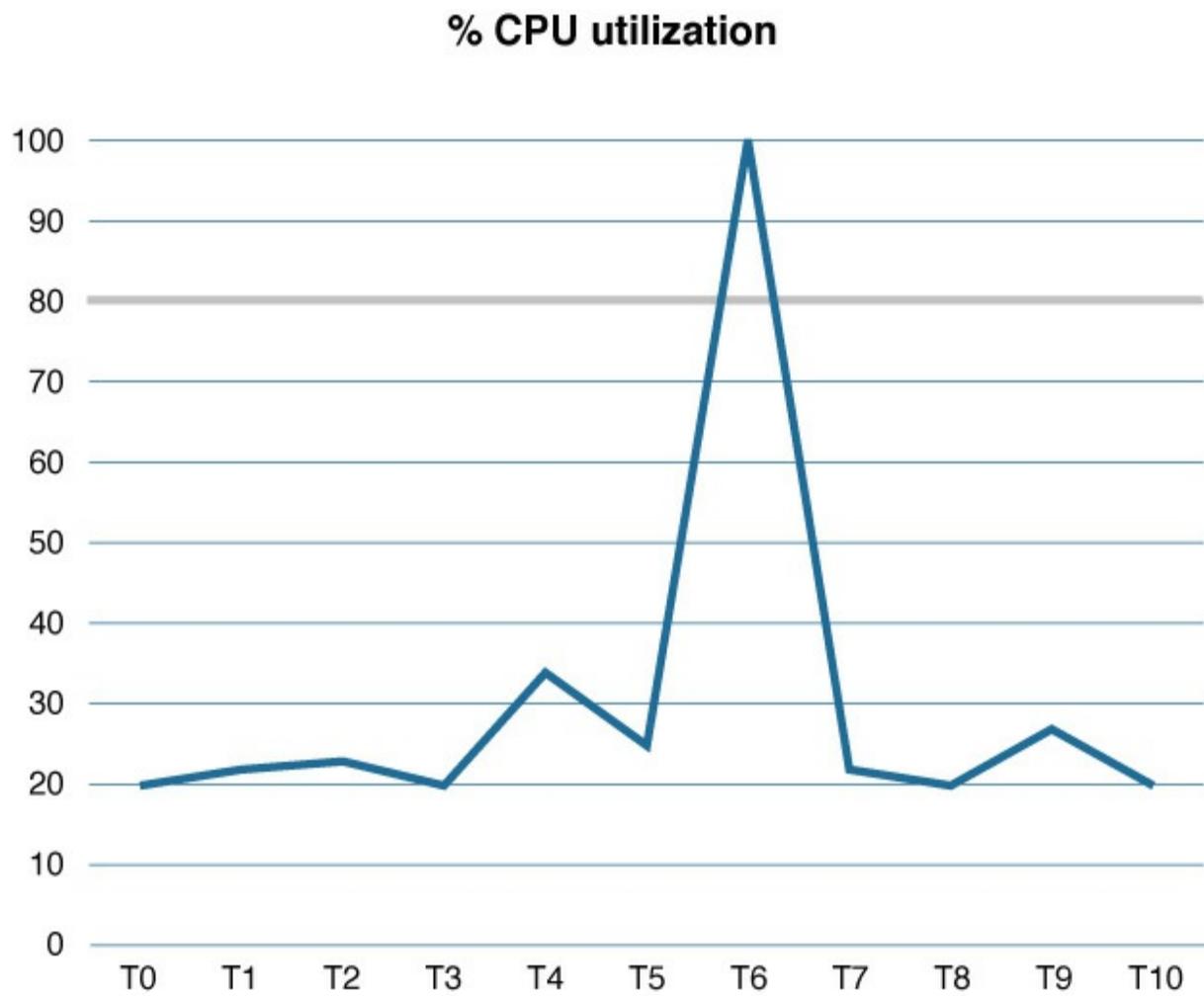


Figure 3-1 % CPU Utilization Graph per Time Measure T

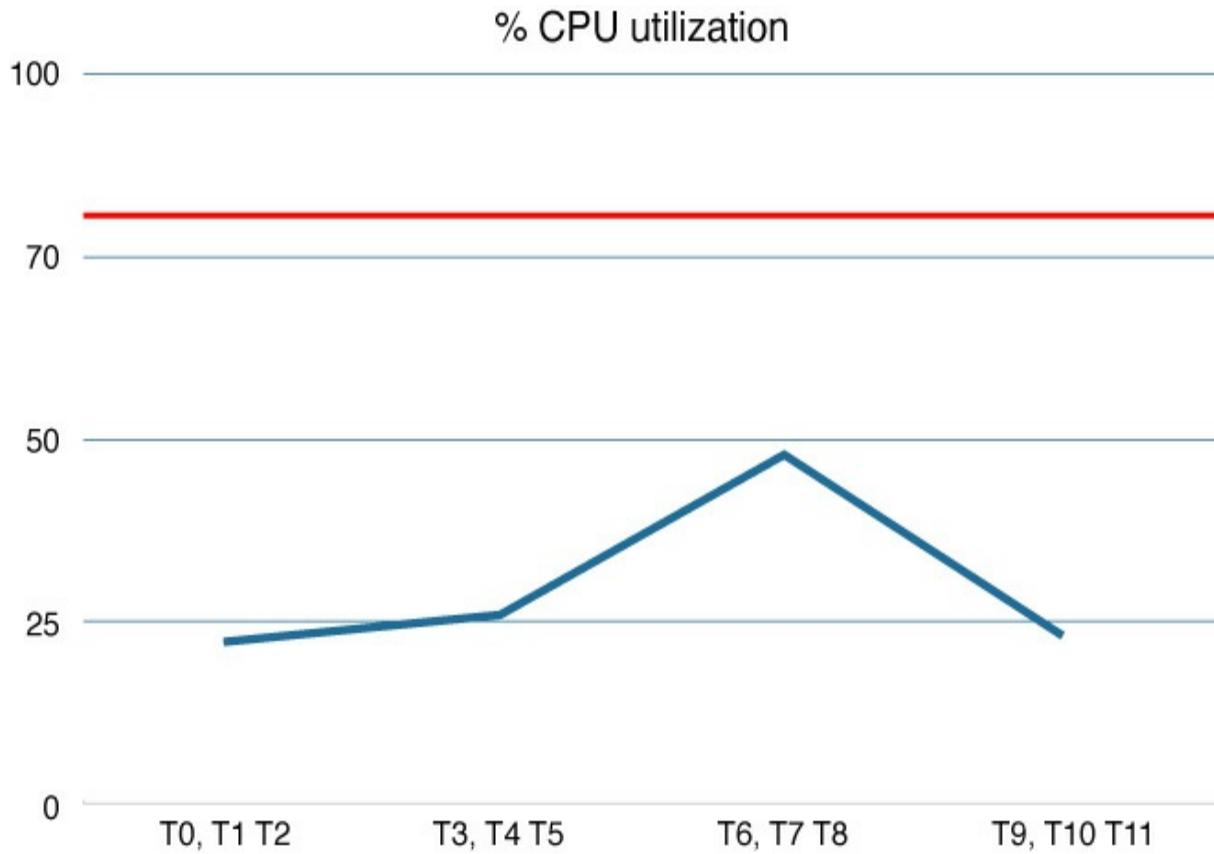


Figure 3-2 % CPU Utilization Graph per Aggregated Time Measure T

In the monitoring use case, it is typical to monitor using aggregated results at time intervals (for example, 15 or 30 seconds). If the aggregated result is over a defined CPU utilization threshold, it is a more accurate metric to act on. Tools like Kibana support aggregation natively.

As another example of aggregating data from multiple sources to achieve better insights, consider the following scenario: You have two interfaces connecting the same two devices, and you are monitoring all interfaces for bandwidth utilization. Your monitoring tool has a defined threshold for bandwidth utilization percentage and automatically provisions a new interface if the threshold is reached. For some reason, most of your traffic is taking one of the interfaces, which triggers your monitoring tool's threshold. However, you still have the other interface bandwidth available. A more accurate aggregated metric would be the combined bandwidth available for the path (an aggregate of the data on both interfaces).

Finally, in some cases, you can aggregate logs with the addition of a

quantifier instead of repeated a number of times—although this is often out of your control because many tools either do not support this feature or apply it automatically. This type of aggregation can occur either in the device producing the logs or on the log collector. It can be seen as a compression technique as well (see [Example 3-7](#)). This type of aggregation is something to have in mind when analyzing data rather than something that you configure.

Example 3-7 Log Aggregation on Cisco’s NX-OS

```
Original:  
2020 Dec 24 09:34:11 N7009 %VSHD-5-VSHD_SYSLOG_CONFIG_I: Configur  
2020 Dec 24 09:35:09 N7009 %VSHD-5-VSHD_SYSLOG_CONFIG_I: Configur  
Aggregated:  
2020 Dec 24 09:34:11 N7009 %VSHD-5-VSHD_SYSLOG_CONFIG_I: Configur  
2020 Dec 24 09:35:09 N7009 last message repeated 1 time
```

Data Visualization

Humans are very visual. A good visualization can illustrate a condition even if the audience does not have technical knowledge. This is especially helpful for alarming conditions. For example, you do not need to understand that the CPU for a device should not be over 90% if there is a red blinking light over it.

Visualizing data can help you build better network automation solutions or even perceive the need for an automated solution. However, automation solutions can also build data visualization artifacts.

There are a number of tools that can help you visualize data. For example, [Chapter 1, “Types of Network Automation,”](#) discusses Kibana, which can be used for log data; Grafana, which can be used for metric data; and Splunk, which can be used for most data types. There are also data visualization tools specific for NetFlow or IPFIX data; for example, Cisco’s Tetration. This type of flow data is typically exported to specialized tools to allow you to visualize who the endpoints are talking to, what protocols they are using, and what ports they are using.

Tip

When you automate actions, base the actions on data rather than on assumptions.

Say that you want to visualize the CPU utilization of routers. The best data gathering method in this scenario would be model-driven telemetry (but if telemetry is not available, SNMP can also work), and Grafana would be a good visualization choice. Grafana integrates with all kinds of databases, but for this type of time-series data, InfluxDB is a good fit. The configuration steps would be as follows:

Step 1. Configure a telemetry subscription in the router.

Step 2. Configure a telemetry server (for example, Telegraf) to listen in.

Step 3. Store the data in a database (for example, InfluxDB).

Step 4. Configure Grafana's data source as the database table.

The resulting visualization might look like the one in [Figure 3-3](#), where you can see the millicores on the vertical axis and the time on the horizontal axis.

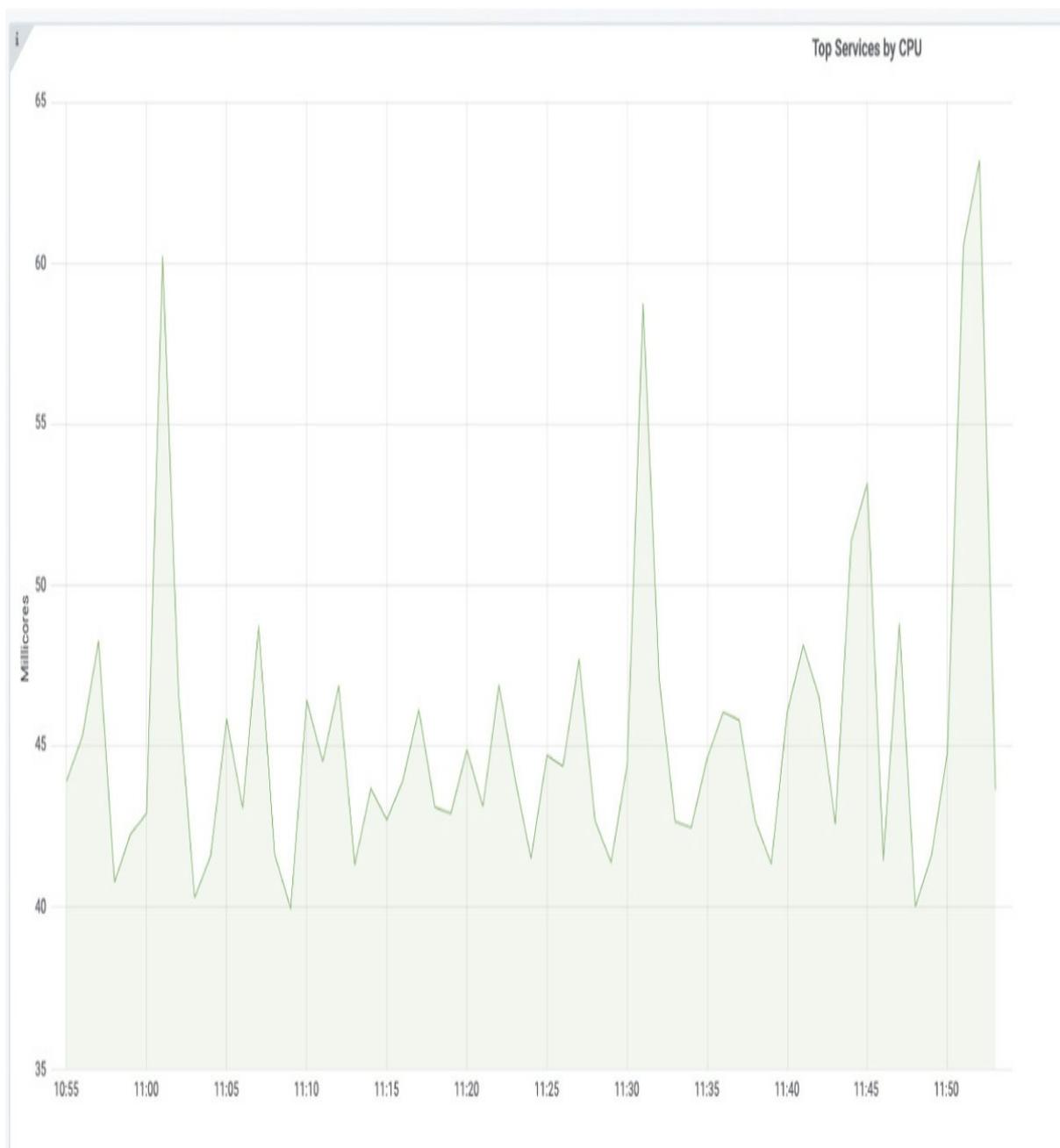


Figure 3-3 *Grafana Displaying a Router CPU Utilization Graph per Minute*

This type of data visualization could alert you to the need for an automated solution to combat CPU spikes, for example, if they are occurring frequently and impacting service. Many times, it is difficult to perceive a need until you see data represented in a visual format.

Although data visualization is a discipline on its own and we only touch on a

very small part of it, there are a couple things to have in mind when creating visualizations. For one thing, you should choose metrics that clearly represent the underlying status. For example, for CPU utilization, typically the percentage is shown and not cores or threads. This is because interpreting a percentage is much easier than knowing how many cores are available on a specific device and contrasting that information with the devices being used. On the other hand, when we are looking at memory, most of the time we represent it with a storage metric (such as GB or MB) rather than a percentage. Consider that 10% of memory left can mean a lot in a server with 100 TB but very little in a server with 1 TB; therefore, representing this metric as 10 TB left or 100 GB left would be more illustrative.

Choosing the right metric scale can be challenging. [Table 3-3](#) illustrates commonly used scales for enterprise network component visualizations.

Table 3-3 *Data Visualization Scales*

Metric	Scale
CPU utilization	Percentage over time
Memory utilization	Specific storage in time (for example, GB)
Disk operations	Operations over time
Network traffic	Specific transmission over time (for example Gbps)

Another important decision is visualization type. There are many commonly used visualizations. The following types are commonly used in networking:

- Line charts
- Stat panels/gauges
- Bar charts

Your choice of visualization will mostly come down to two answers:

- Are you trying to compare several things?
- Do you need historical context?

The answer to the first question indicates whether you need multiple- or single-line visualizations. For example, if you want to visualize the memory usage of all components within a system, you could plot it as shown in [Figure](#)

[3-4](#). This line chart with multiple lines would allow you to compare the components and get a broader perspective. In this figure, you can see that the app-hosting application is consuming more than twice as much memory as any other running application.

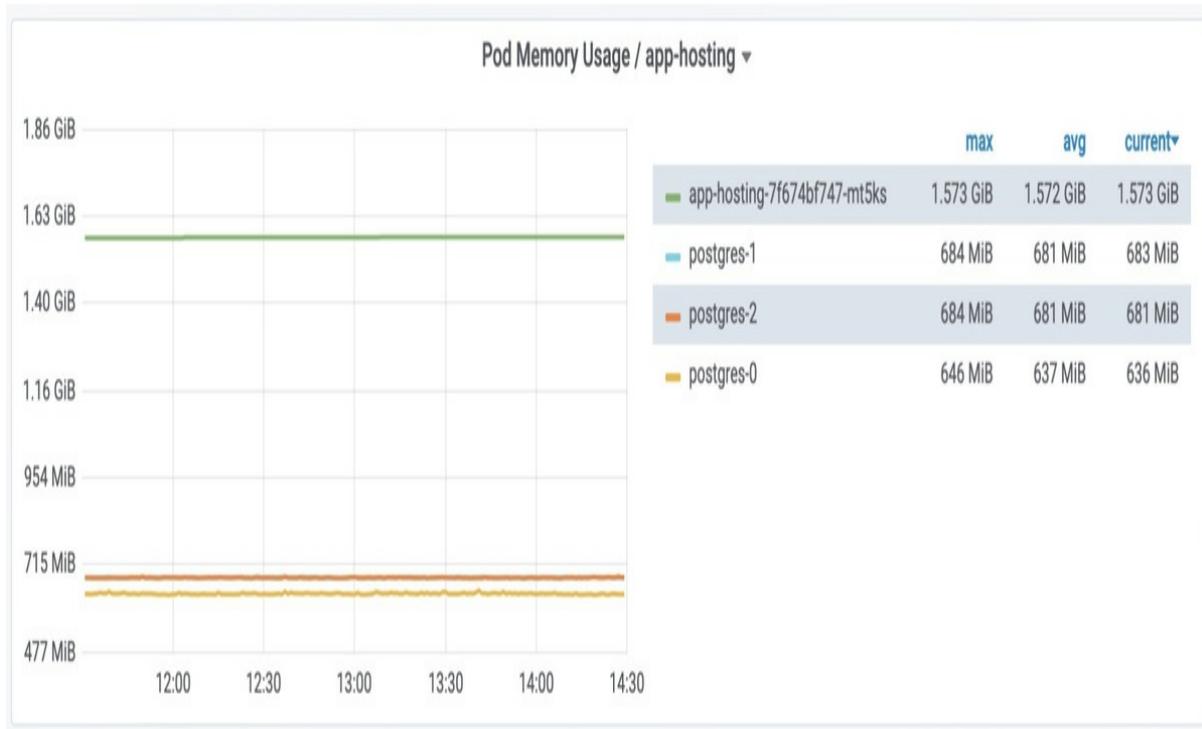


Figure 3-4 Virtual Memory Utilization Graph of All Components in a Container

If you simply want to see the memory usage for a single component, such as a router or an application running on a system, you do not need to pollute your visualization with unneeded information that would cause confusion without adding value. In this case, a single-line chart like the one in [Figure 3-5](#) would suffice.



Figure 3-5 Virtual Memory Utilization Graph of Sensor Assurance Application

Both [Figures 3-4](#) and [3-5](#) give you a chronological view of the metric being measured. If you do not need historical context, and you just want to know the status at the exact moment, you can use panels or bar charts instead. For example, [Figure 3-6](#) shows a point-in-time view of the [Figure 3-4](#) metrics.

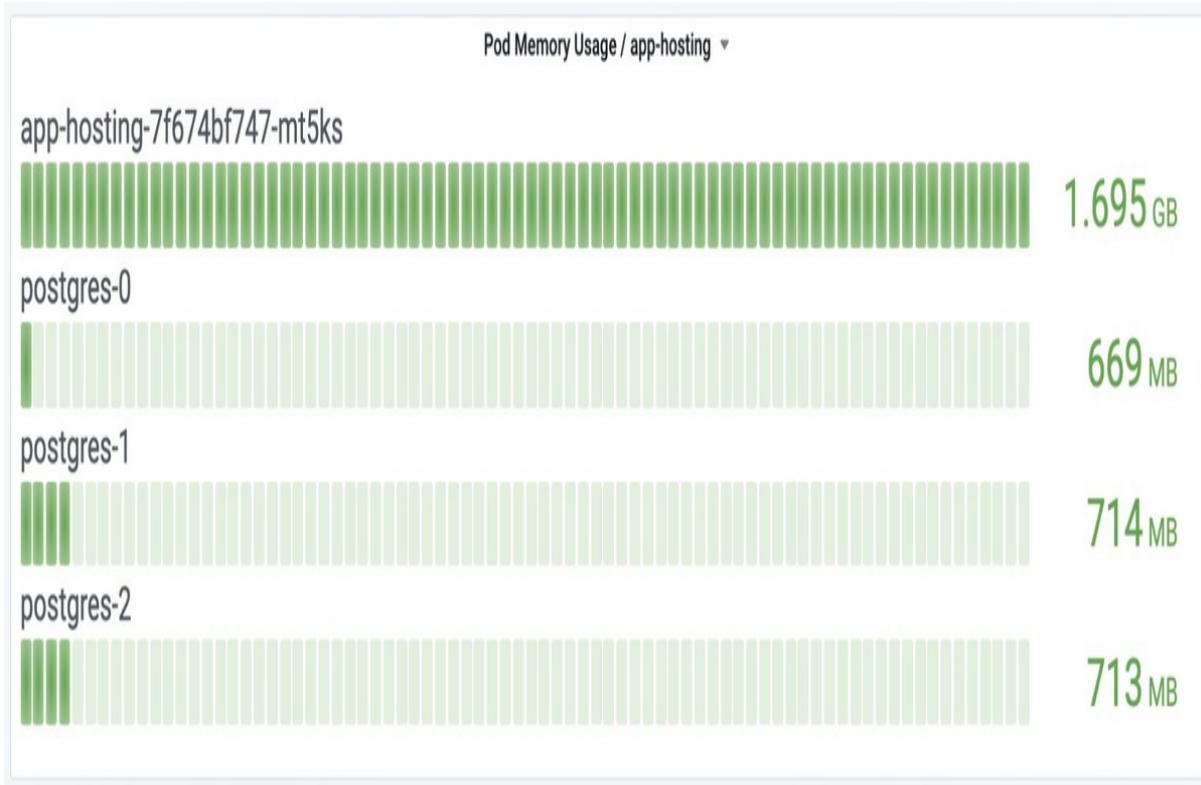


Figure 3-6 Current Virtual Memory Utilization Bar Chart of All Components in a Container

When you do not need a chronological understanding of a metric and you have a single component to represent, the most commonly used visualization is a gauge or a stat panel. [Figure 3-7](#) shows a point-in-time view of the [Figure 3-5](#) metric.

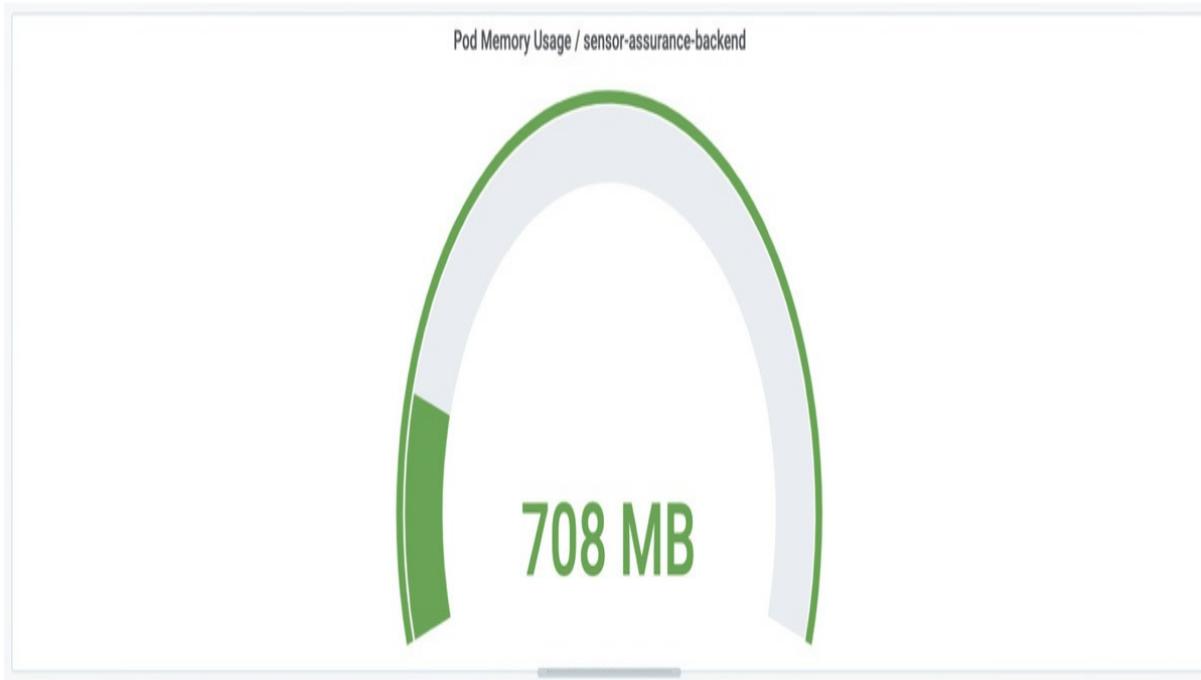


Figure 3-7 Gauge Representation of the Current Virtual Memory Utilization of Sensor Assurance Application

A way to further enhance visualization is to use color. You can't replace good visualization metrics and types with color. However, color can enhance good visualization metrics and types. Color thresholds in visualizations can enable quicker insights. For example, by defining 80% memory utilization or higher as red, 60% to 79% utilization as yellow, and 59% or less as green, a visualization can help an operator get a feeling for the overall status of a system with a simple glance at a dashboard.

In some cases, you might want to automate a data visualization instead of using a prebuilt tool. This might be necessary if your use case is atypical, and market tools do not fulfill it. For example, say that a customer has an ever-growing and ever-changing network. The network engineers add devices, move devices across branches, and remove devices every day. The engineering team wants to have an updated view of the topology when requested. The team is not aware of any tool on the market that can achieve this business need, so it decides to build a tool from scratch. The team uses Python to build an automated solution hosted on a central station and that consists of a tool that collects CDP neighbor data from devices, cross-references the data between all devices, and uses the result to construct

network diagrams. This tool can be run at any time to get an updated view of the network topology.

Note

Normally, you do not want to build a tool from scratch. Using tools that are already available can save you time and money.

Data Insights

An insight, as mentioned earlier, is knowledge gained from analyzing information. The goal of gathering data is to derive insights. How do you get insights from your data? In two steps:

Step 1. You need to build a knowledge base. Knowing what is expected behavior and what is erroneous behavior is critical when it comes to understanding data. This is where you, the expert, play a role. You define what is good and what is bad. In addition, new techniques can use the data itself to map what is expected behavior and what is not. We touch on these techniques later in this section.

Step 2. You apply the knowledge base to the data—hopefully automatically, as this is a network automation book.

As an example of this two-step process, say that you define that if a device is running at 99% CPU utilization, this is a bad sign (step 1). By monitoring your devices and gathering CPU utilization data, you identify that a device is running at 100% utilization (step 2) and replace it. The data insight here is that a device was not working as expected.

Alarms

Raising an alarm is an action on the insight (in this case, the insight that it's necessary to raise the alarm). You can trigger alarms based on different types of data, such as log data or metrics.

You can use multiple tools to generate alarms (refer to [Chapter 1](#)); the

important parameter is what to alarm on. For example, say that you are gathering metrics from your network using telemetry, and among these metrics is memory utilization. What value for memory utilization should raise an alarm? There is no universal answer; it depends on the type of system and the load it processes. For some systems, running on 90% memory utilization is common; for others, 90% would indicate a problem. Defining the alarm value is referred to as *defining a threshold*. Most of the time, you use your experience to come up with a value.

You can also define a threshold without having to choose a critical value. This technique, called baselining, determines expected behavior based on historical data.

A very simple baselining technique could be using the average from a specific time frame. However, there are also very complex techniques, such as using neural networks. Some tools (for example, Cisco's DNA Center) have incorporated baselining modules that help you set up thresholds.

If you are already using Grafana, creating an alarm is very simple. By editing the [Figure 3-4](#) dashboard as shown in [Figure 3-8](#), you can define the alarm metric. You can use simple metrics such as going over a determined value, or calculated metrics such as the average over a predefined time. [Figure 3-8](#) shows a monitoring graph of disk I/O operations on several services; it is set to alarm if the value reaches 55 MBs.

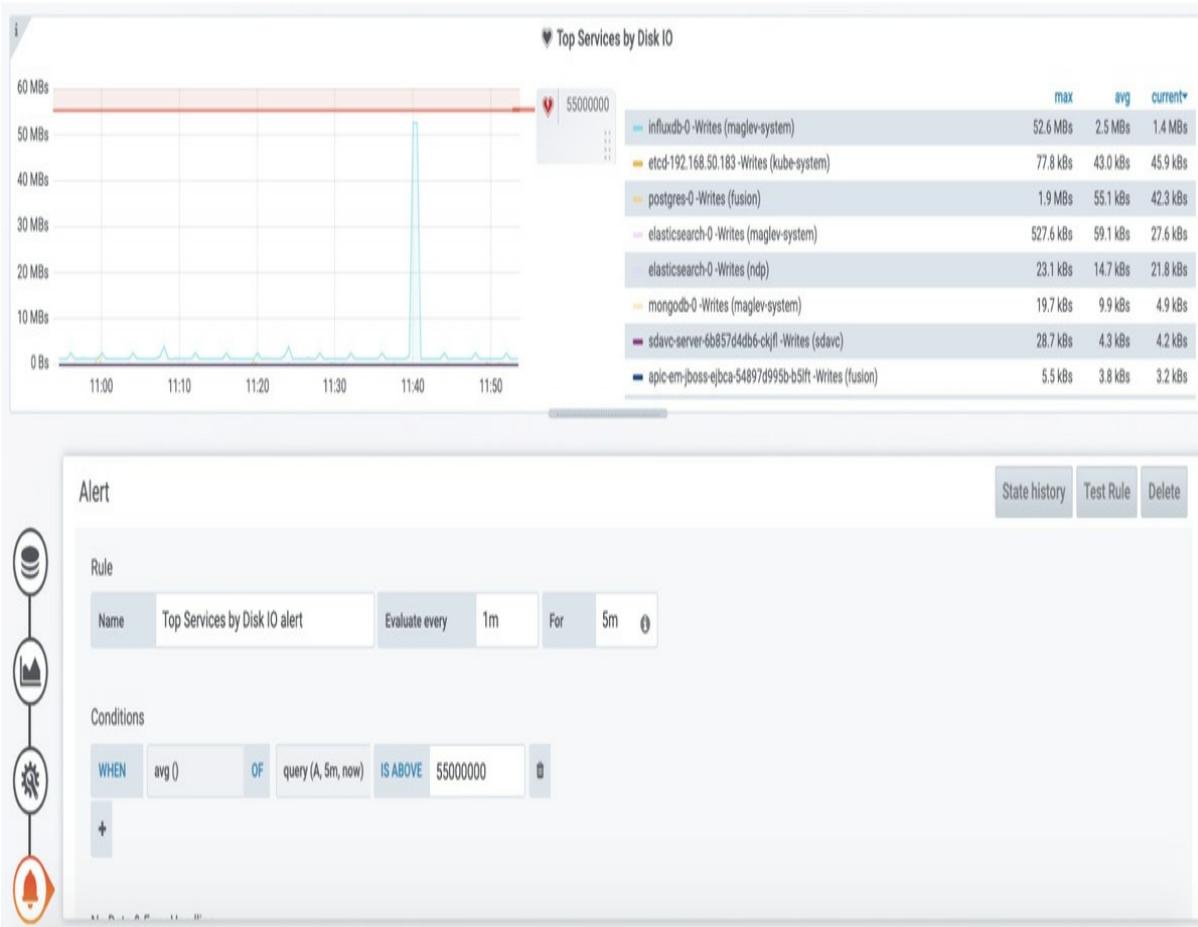


Figure 3-8 Setting Up a Grafana Alarm for 55 MBs Disk I/O

Tools like Kibana also allow you to set up alerts based on log data. For example, [Figure 3-9](#) shows the setup of an alarm based on receiving more than 75 Syslog messages from a database with the error severity in the past 5 minutes.

The screenshot shows the Kibana interface with a log stream on the left and an open 'Create alert' dialog on the right.

Log Stream (Left):

- Stream:** event.dataset
- Search bar:** Search for log entries... (e.g. host.name:host-1)
- Message:**
 - Jan 3, 2021 12:14:05.144 postgresql.log [postgresql.log][LOG] 2021-01-03 11:14:05.144 UTC [534592] med> select order0_.id as col_0_0_, order0_.created_at as order0_ left outer join customers customer1_ on order0_.cus
 - 12:14:05.181 postgresql.log [postgresql.log][LOG] 2021-01-03 11:14:05.181 UTC [540553] ELECT SUM("total_revenue"), SUM("total_cost"), SUM("total_p d") AS "order_count", ("opbeans_product"."selling_price" - ("opbeans_orderline"."order_id") * ("opbeans_product"."sell t", (COUNT("opbeans_orderline"."order_id") * "opbeans_produ s_orderline"."order_id") * "opbeans_product"."cost") AS "to ns_orderline" ON ("opbeans_product"."id" = "opbeans_orderli beans_product"."selling_price" - "opbeans_product"."cost"))
 - 12:14:05.192 postgresql.log [postgresql.log][LOG] 2021-01-03 11:14:05.192 UTC [540553] LECT COUNT(*) AS "..._count" FROM "opbeans_product"
 - 12:14:05.193 postgresql.log [postgresql.log][LOG] 2021-01-03 11:14:05.193 UTC [540553] LECT COUNT(*) AS "..._count" FROM "opbeans_customer"
 - 12:14:05.207 postgresql.log [postgresql.log][LOG] 2021-01-03 11:14:05.207 UTC [540553] LECT COUNT(*) AS "..._count" FROM "opbeans_order"
 - 12:14:05.277 postgresql.log [postgresql.log][LOG] 2021-01-03 11:14:05.277 UTC [540553]

Create alert Dialog (Right):

- Name:** error logs
- Tags (optional):** (empty)
- Check every:** 1 minute
- Notify every:** 1 minute
- Log threshold:**
 - WHEN THE count OF LOG ENTRIES
 - WITH log.level IS error
 - IS more than 75
 - FOR THE LAST 5 minutes
 - GROUP BY Nothing (ungrouped)
- Buttons:** Cancel, Save

Figure 3-9 Setting Up a Kibana Alarm for Syslog Data

In addition to acting as alerts for humans, alarms can be automation triggers; that is, they can trigger automated preventive or corrective actions. Consider the earlier scenario of abnormally high CPU utilization percentage. In this case, a possible alarm could be a webhook that triggers an Ansible playbook to clean up known CPU-consuming processes.

Note

A webhook is a “reverse API” that provides a way for an application to notify about a behavior using HTTP. Some tools are automatically able to receive webhooks and trigger behaviors based on them (for example, RedHat Ansible Tower). You can also build your own webhook receivers and call actions, depending on the payload.

[Figure 3-10](#) illustrates an attacker sending CPU punted packets to a router, which leads to a spike in CPU usage. This router is configured to send telemetry data to Telegraf, which stores it in an InfluxDB database. As Grafana ingests this data, it notices the unusual metric and triggers a configured alarm that uses an Ansible Tower webhook to run a playbook and configure an ACL that will drop packets from the attacker, mitigating the effects of the attack.

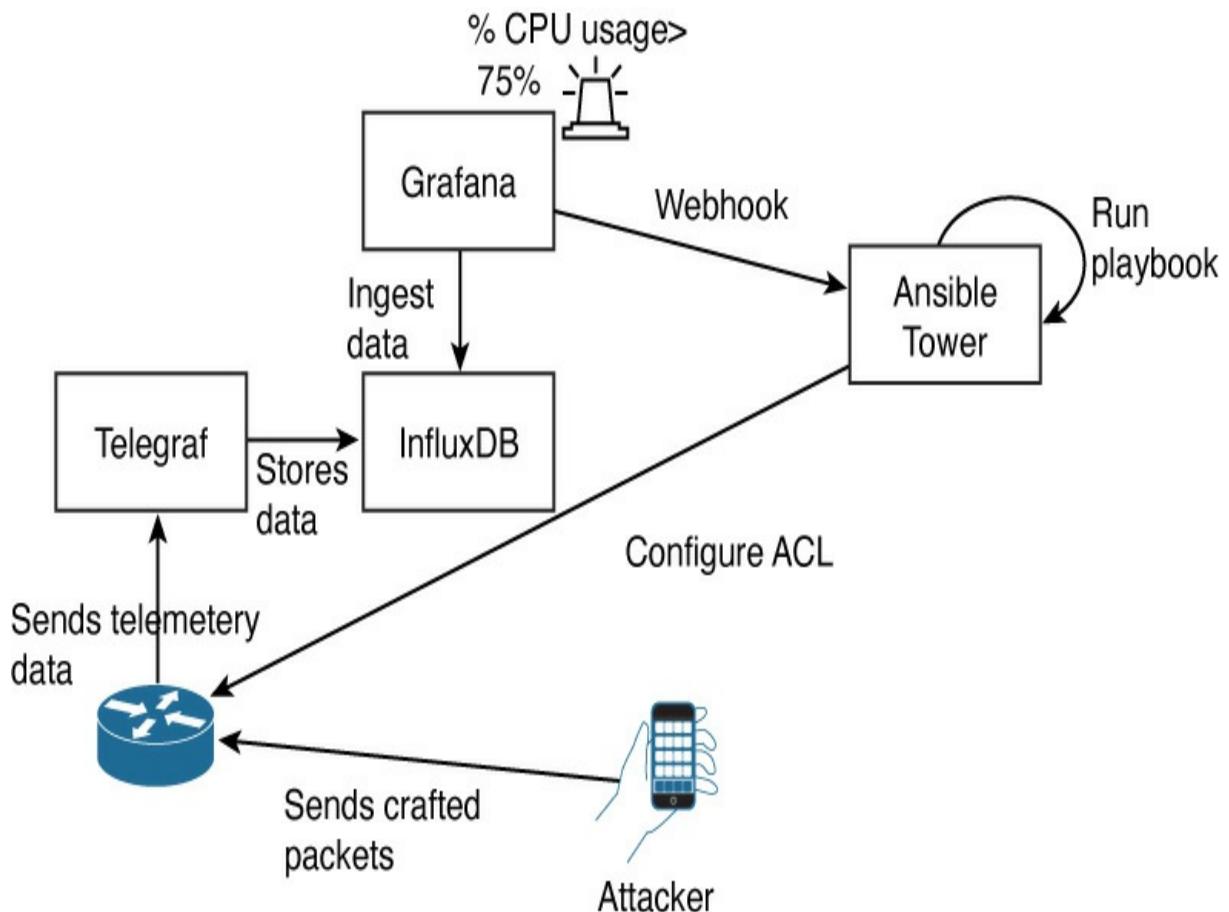


Figure 3-10 Automated Resolution Triggered by Webhook

Typically, deploying a combination of automated action and human notification is the best choice. Multiple targets are possible when you set up an alarm to alert humans, such as email, SMS, dashboards, and chatbots (for example, Slack or Webex Teams).

In summary, you can use an alarm to trigger actions based on a state or an event. These actions can be automated or can require manual intervention.

The way you set up alarms and the actions available on alarms are tightly coupled with the tools you use in your environment. Grafana and Kibana are widely used in the industry, and others are available as well, including as Splunk, SolarWinds, and Cisco's DNA Center.

Configuration Drift

In [Chapter 1](#), we touched on is the topic of configuration drift. If you have worked in networking, you know it happens. Very few companies have enough controls in place to completely prevent it, especially if they have long-running networks that have been in place more than 5 years. So, how do you address drift?

You can monitor device configurations and compare them to known good ones (templates). This tells you whether configuration drift has occurred. If it has, you can either replace the device configuration with the template or the other way around. The decision you make depends on what is changed.

You can apply templates manually, but if your network has a couple hundred or even thousands of devices, it will quickly become a burden. Ansible can help with this task, as shown in [Example 3-8](#).

Note

Do not worry if you don't fully understand the code in this example yet. It is meant to function as a future refence, and you might want to circle back here after you read [Chapters 4](#) and [5](#).

Example 3-8 Using Ansible to Identify Configuration Differences in a Switch

```
$ cat config_dif.yml
---
- hosts: all
  gather_facts: no
  tasks:
    - name: diff against the startup config
      cisco.ios.ios_config:
```

```

diff_against: intended
intended_config: "{{ lookup('file', 'template.txt') }}"

$ cat inv.yaml
all:
  hosts:
    switch_1:
      ansible_host: "10.10.10.1"
      ansible_network_os: cisco.ios.ios
  vars:
    ansible_connection: ansible.netcommon.network_cli
    ansible_user: "cisco"
    ansible_password: "C!sc0123"

$ ansible-playbook -i inv.yaml config_dif.yml -diff
PLAY [all] ****
TASK [diff against the startup config]
*****
ok: [switch_1]

PLAY RECAP ****
switch_1 : ok=1     changed=0     unreachable=0
rescued=0   ignored=0

Modified the ssh version to 1 instead of 2 on the template file
(template.txt)

$ cat template.txt
#output omitted#
ip ssh version 1
#output omitted#

$ ansible-playbook -i inv.yaml config_dif.yml --diff
PLAY [all] ****
TASK [diff against the startup config]
*****
--- before

```

```
+++ after
@@ -253,7 +253,7 @@
no ip http secure-server
ip route vrf Mgmt-vrf 0.0.0.0 0.0.0.0 3.6.0.254 name LAB-MGMT-GW
ip tacacs source-interface Vlan41
-ip ssh version 2
+ip ssh version 1
ip scp server enable
tacacs-server host 192.168.48.165
tacacs-server host 1.1.1.1

changed: [switch_1]

PLAY RECAP ****
switch_1 : ok=1    changed=1    unreachable=0
rescued=0  ignored=0
```

[Example 3-8](#) shows a template file (template.txt) with the expected configuration, which is the same configuration initially used on the switch. The example shows how you can connect automatically to the host to verify whether its configuration matches the provided template. If it does not, you see the differences in the output (indicated with the + or – sign). The example shows that, on the second execution, after the template is modified, the output shows the differences.

Tip

Something to keep in mind when comparing configurations is that you must ignore hashes.

You can also achieve configuration compliance checking by using the same type of tool and logic but, instead of checking for differences against a template, checking for deviations from the compliance standards (for example, using SHA-512 instead of SHA-256).

AI/ML Predictions

Insights can come from artificial intelligent (AI) or machine learning (ML) techniques. AI and ML have been applied extensively in the past few years in many contexts (for example, for financial fraud detection, image recognition, and natural language processing). They can also play a role in networking.

ML involves constructing algorithms and models that can learn to make decisions/predictions directly from data without following predefined rules. Currently, ML algorithms can be divided into three major categories: supervised, unsupervised, and reinforcement learning. Here we focus on the first two categories because reinforcement learning is about training agents to take actions, and that type of ML is very rarely applied to networking use cases.

Supervised algorithms are typically used for classification and regression tasks, based on *labeled data* (where labels are the expected result). Classification involves predicting a result from a predefined set (for example, predicting malicious out of the possibilities malicious or benign). Regression involves predicting a value (for example, predicting how many battery cycles a router will live through).

Unsupervised algorithms try to group data into related clusters. For example, given a set of NetFlow log data, grouping it with the intent of trying to identify malicious flows would be considered unsupervised learning.

On the other hand, given a set of NetFlow log data where you have previously identified which subset of flows are malicious and using it to whether if future flows are malicious would be considered supervised learning.

One type is not better than the other; supervised and unsupervised learning aim to address different types of situations.

There are three major ways you can use machine learning:

- Retraining models
- Training your own models with automated machine learning (AutoML)
- Training your models manually

Before you learn about each of these ways, you need to understand the steps

involved and training and using a model:

Step 1. Define the problem (for example, prediction, regression, clustering).

You need to define the problem you are trying to solve, the data that you need to solve it, and possible algorithms to use.

Step 2. Gather data (for example, API, syslog, telemetry). Typically you need a lot of data, and gathering data can take weeks or months.

Step 3. Prepare the data (for example, parsing, aggregation). There is a whole discipline called data engineering that tries to accomplish the most with data in a machine learning context.

Step 4. Train the model. This is often a trial-and-error adventure, involving different algorithms and architectures.

Step 5. Test the model. The resulting models need to be tested with sample data sets to see how they perform at predicting the problem you defined previously.

Step 6. Deploy/use the model. Depending on the results of testing, you might have to repeat steps 4 and 5; when the results are satisfactory, you can finally deploy and use the model.

These steps can take a long time to work through, and the process can be expensive. The process also seems complicated, doesn't it? What if someone already addressed the problem you are trying to solve? These is where pretrained models come in. You find these models inside products and in cloud marketplaces. These models have been trained and are ready to be used so you can skip directly to step 6. The major inconvenience with using pretrained models is that if your data is very different from what the model was trained with, the results will not be ideal. For example, a model for detecting whether people were wearing face masks was trained with images at 4K resolution. When predictions were made with CCTV footage (at very low resolution), there was a high false positive rate. Although it may provide inaccurate results, using pretrained models is typically the quickest way to get machine learning insights in a network.

If you are curious and want to try a pretrained model, check out Amazon Rekognition, which is a service that identifies objects within images. Another example is Amazon Monitron, a service with which you must install

Amazon-provided sensors that analyze industrial machinery behavior and alert you when they detect anomalies.

AutoML goes a step beyond pretrained models. There are tools available that allow you to train your own model, using your own data but with minimal machine learning knowledge. You need to provide minimal inputs. You typically have to provide the data that you want to use and the problem you are trying to solve. AutoML tools prepare the data as they see fit and train several models. These tools present the best-performing models to you, and you can then use them to make predictions.

With AutoML, you skip steps 3 through 5, which is where the most AI/ML knowledge is required. AutoML is commonly available from cloud providers.

Finally, you can train models in-house. This option requires more knowledge than the other options. Python is the tool most commonly used for training models in-house. When you choose this option, all the steps apply.

An interesting use case of machine learning that may spark your imagination is in regard to log data. When something goes wrong—for example, an access switch reboots—you may receive many different logs from many different components, from routing neighborships going down to applications losing connectivity and so on. However, the real problem is that a switch is not active. Machine learning can detect that many of those logs are consequences of a problem and not the actual problem, and it can group them accordingly. This is part of a new discipline called AIOps (artificial intelligence operations). A tool not mentioned in [Chapter 1](#) that tries to achieve AIOps is Moogsoft.

Here are a couple examples of machine learning applications for networks:

- Malicious traffic classification based on k-means clustering.
- Interface bandwidth saturation forecast based on recurrent neural networks.
- Log correlation based on natural language processing.
- Traffic steering based on time-series forecasting.

The code required to implement machine learning models is relatively simple, and we don't cover it in this book. The majority of the complexity is in gathering data and parsing it. To see how simple using machine learning

can be, assume that data has already been gathered and transformed and examine [Example 3-9](#), where x is your data, and y is your labels (that is, the expected result). The first three lines create a linear regression model and train it to fit your data. The fourth line makes predictions. You can pass a value (new_value , in the same format as the training data, x), and the module will try to predict its label.

Example 3-9 Using Python’s *sklearn* Module for Linear Regression

```
from sklearn.linear_model import LinearRegression

regression_model = LinearRegression()
regression_model.fit(x, y)
y_predicted = regression_model.predict(new_value)
```

Case Studies

This section describes three case studies based on real customer situations. It examines, at a high level, the challenges these companies had and how we they addressed them with the network automation techniques described throughout this book. These case studies give you a good idea of the benefits of automation in real-world scenarios.

Creating a Machine Learning Model with Raw Data

Raw data is data in the state in which it was collected from the network—that is, without transformations. You have seen in this chapter that preparing data is an important step in order to be able to derive insights. A customer did not understand the importance of preparing data, and this is what happened.

Company XYZ has a sensitive business that is prone to cyberattacks. It had deployed several security solutions, but because it had machine learning experts, XYZ wanted to enhance its current portfolio with an in-house solution.

The initial step taken was to decide which part of the network was going to be used, and XYZ decided that the data center was the best fit. XYZ decided that it wanted to predict network intrusion attacks based on flow data from network traffic.

After careful consideration, the company started collecting flow data from the network by using IPFIX. [Table 3-4](#) shows the flow's format.

Table 3-4 Flow Data Fields

Duration	Protocol	Src IP	Src Port	Dst IP	Dst Port	Packets	Bytes	Flows	Flags
813	TCP	10.0.0.2	56979	10.0.0.3	8080	12024	10300	1	AP

After collecting data for a couple months, a team manually labeled each collected flow as suspicious or normal.

With the data labeled, the machine learning team created a model, using a decision tree classifier.

The results were poor, with accuracy in the 81% range. We were engaged and started investigating.

The results from the investigation were clear: XYZ had trained the model with all features in their raw state—so the data was unaltered from the time it was collected, without any data transformation. Bytes values appeared in different formats (sometimes 1000 bytes and sometimes 1 KB), and similar problems occurred with the number of packets. Another clear misstep was that the training sample had an overwhelming majority of normal traffic compared to suspicious traffic. This type of class imbalance can damage the training of a machine learning model.

We retrained the model, this time with some feature engineering (that is, data preparation techniques). We separated each flag into a feature of its own, scaled the data set in order for the classes to be more balanced, and changed data fields so they were in the same units of measure (for example, all flows in KB), among other techniques.

Our changes achieved 95.9% accuracy.

This case study illustrates how much difference treating the data can make.

The ability to derive insights from data, with or without machine learning, is mostly influenced by the data quality. We trained exactly the same algorithm the original team used and got a 14% improvement just by massaging the data.

How a Data Center Reduced Its Mean Time to Repair

Mean time to repair (MTTR) is a metric that reflects the average time taken to troubleshoot and repair a failed piece of equipment or component.

Customer ABC runs critical services in its data center, and outages in those services incur financial damages along with damages to ABC's brand reputation. ABC owns a very complex data center topology with a mix of platforms, vendors, and versions. All of its services are highly redundant due to the fact that the company's MTTR was high.

We were involved to try to drive down the MTTR as doing so could have a big impact on ABC's business operations. The main reason we could pinpoint as causing the high MTTR metric was the amount of knowledge needed to triage and troubleshoot any situation. In an ecosystem with so many different vendors and platforms, data was really sparse.

We developed an automation solution using Python and natural language processing (NLP) that correlated logs across these platforms and deduplicated them. This really enabled ABC's engineers to understand the logs in a common language.

In addition, we used Ansible playbooks to apply configurations after a device was replaced. The initial workflow consisted of replacing faulty devices with new ones and manually configuring them using the configuration of the replaced device. This process was slow and error prone.

Now after a device is identified for replacement at ABC, the following workflow occurs:

- Step 1.** Collect the configuration of the faulty device (using Ansible).
- Step 2.** Collect information about endpoints connected to the faulty device (using Ansible).

Step 3. Manually connect/cable the new device (manually).

Step 4. Assign the new device a management connection (manually).

Step 5. Configure the new device (using Ansible).

Step 6. Validate that the endpoint information from the faulty device from Step 2 matches (using Ansible).

This new approach allows ABC to ensure that the configurations are consistent when devices are replaced and applied in a quick manner. Furthermore, it provides assurance regarding the number and type of connected endpoints, which should be the same before and after the replacement.

In [Chapter 5](#), you will see how to create your own playbooks to collect information and configure network devices. In the case of ABC, two playbooks were used: one to retrieve configuration and endpoint information from the faulty device and store it and another to configure and validate the replacement device using the previously collected information. These playbooks are executed manually from a central server.

Both solutions together greatly reduced the time ABC takes to repair network components. It is important to note that this customer replaced faulty devices with similar ones, in terms of hardware and software. If a replacement involves a different type of hardware or software, some of the collected configurations might need to be altered before they can be applied.

Network Migrations at an Electricity Provider

A large electricity provider was responsible for a whole country. It had gone through a series of campus network migrations to a new technology, but after a couple months, it had experienced several outages. This led to the decision to remove this new technology, which entailed yet another series of migrations. These migrations had a twist: They had to be done in a much shorter period of time due to customer sentiment and possibility of outages while on the current technology stack.

Several automations were applied, mostly for validation, as the migration activities entailed physical moves, and automating configuration steps wasn't

possible.

The buildings at the electricity provider had many floors, and the floors had many devices, including the following:

- IP cameras
- Badge readers
- IP phones
- Desktops
- Building management systems (such as heating, air, and water)

The procedure consisted of four steps:

Step 1. Isolate the device from the network (by unplugging its uplinks).

Step 2. Change the device software version.

Step 3. Change the device configuration.

Step 4. Plug in the device to a parallel network segment.

The maintenance windows were short—around 6 hours—but the go-ahead or rollback decision had to be made at least 3 hours after the window start. This was due to the time required to roll back to the previous known good state.

We executed the first migration (a single floor) without any automation. The biggest hurdle we faced was verifying that every single endpoint device that was working before was still working afterwards. (Some devices may have not been working before, and making them work on the new network was considered out of scope.) Some floors had over 1000 endpoints connected, and we did not know which of them were working before. We found ourselves—a team of around 30 people—manually testing these endpoints, typically by making a call from an IP phone or swiping a badge in a reader. It was a terrible all-nighter.

After the first migration, we decided to automate verifications. This automation collected many outputs from the network before the migration, including the following:

- CDP neighbors
- ARP table

- MAC address table
- Specific connectivity tests (such as ICMP tests)

Having this information on what was working before enabled us to save tens of person-hours.

We parsed this data into tables. After devices were migrated (manually), we ran the collection mechanisms again. We therefore had a picture of the before and a picture of the after. From there, it was easy to tell which endpoint devices were working before and not working afterward, so we could act on only those devices.

We developed the scripts in Python instead of Ansible. The reason for this choice was the team's expertise. It would be possible to achieve the same with Ansible. (Refer to [Example 1-9](#) in [Chapter 1](#) for a partial snippet of this script.)

There were many migration windows for this project. The first 8-hour maintenance window was dedicated to a single floor. In our the 8-hour window, we successfully migrated and verified six floors.

The time savings due to automating the validations were a crucial part of the success of the project. The stress reduction was substantial as well. At the end of each migration window, the team was able to leave, knowing that things were working instead of fearing being called the next morning for nonworking endpoints that were left unvalidated.

Summary

This chapter covers what to do after you have collected data. It explains common data preparation methods, such as parsing and aggregation, and discusses how helpful it can be to visualize data.

In this chapter, you have seen that data by itself has little value, but insights derived from high-quality data can be invaluable when it comes to making decisions. In particular, this chapter highlights alarms, configuration drift, and AI/ML techniques.

Finally, this chapter presents real case studies of data automation solutions

put in place to improve business outcomes.

[Chapter 4](#) covers Ansible. You have already seen a few examples of what you can accomplish with Ansible, and after you read [Chapter 4](#), you will understand the components of this tool and how you can use it. Be sure to come back and revisit the examples in this chapter after you master the Ansible concepts in [Chapters 4 and 5](#).

Review Questions

You can find answers to these questions in [Appendix A, “Answers to Review Questions.”](#)

- 1.** If you need to parse log data by using regex, which symbol should you use to match any characters containing the digits 0 through 9?
 - a.** \d
 - b.** \D
 - c.** \s
 - d.** \W

- 2.** If you need to parse log data by using regex, which symbol should you use to match any whitespace characters?
 - a.** \d
 - b.** \D
 - c.** \s
 - d.** \W

- 3.** Which of the IP-like strings match the following regex? (Choose two.)
\d+\.\d+\.\d+\.\d+
 - a.** 255.257.255.255
 - b.** 10.00.0.1

c. 10.0.0.1/24

d. 8.8,8.8

- 4.** You work for a company that has very few network devices but that has grown substantially lately in terms of users. Due to this growth, your network has run at nearly its maximum capacity. A new project is being implemented to monitor your devices and applications. Which technique can you use to reduce the impact of this new project in your infrastructure?
- a.** Data parsing
 - b.** Data aggregation
 - c.** Dara visualization
- 5.** You are building a new Grafana dashboard. Which of the following is the most suitable method for representing a single device's CPU utilization?
- a.** Bar chart
 - b.** Line chart
 - c.** Gauge
 - d.** Pie chart
- 6.** You are building a new Grafana dashboard. Which of the following is the most suitable method for representing the memory utilization of all components in a system?
- a.** Bar chart
 - b.** Line chart
 - c.** Gauge
 - d.** Pie chart
- 7.** True or false: With automated systems, alarms can only trigger human intervention.

- a.** True
 - b.** False
- 8.** Your company asks you to train a machine learning model to identify unexpected logs, because it has been storing logs for the past years. However, you lack machine learning expertise. Which of the following is the most suitable method to achieve a working model?
- a.** Use AutoML
 - b.** Use a pretrained model that you find online
 - c.** Use an in-house trained model
- 9.** In the context of regex, what does the symbol + mean?
- a.** One or more occurrences
 - b.** Zero or more occurrences
 - c.** Ends with
 - d.** Starts with
- 10.** In the context of machine learning, your company wants to identify how many virtual machines it needs to have available at a specific time of day to process transactions. What type of problem is this?
- a.** Regression
 - b.** Classification
 - c.** Clustering
 - d.** Sentiment analysis

Chapter 4. Ansible Basics

You learned a little about Ansible in [Chapter 1, “Types of Network Automation,”](#) and saw some examples of using it in [Chapter 3, “Using Data from Your Network.”](#) Ansible is the number-one tool used in enterprise networking automation. You can use it for most of the use cases covered so far, including configuration, provisioning, monitoring, compliance, interaction, testing, and reporting.

This chapter covers the following facets of Ansible:

- Main characteristics
- Installation
- Concepts such as variables, loops, and conditionals
- Inventory
- Playbooks
- Roles

After you have finished this chapter, you should be able to understand, build, and execute your own playbooks.

Ansible Characteristics

Ansible is agentless and uses SSH to connect to its targets. It has two types of nodes: control and managed node. A control node is a node where you have Ansible installed. It functions as a management station that connects to the managed nodes and executes instructions. Managed nodes are the targets where you want to execute operations (for example, network routers or switches); [Figure 4-1](#) shows a reference architecture.

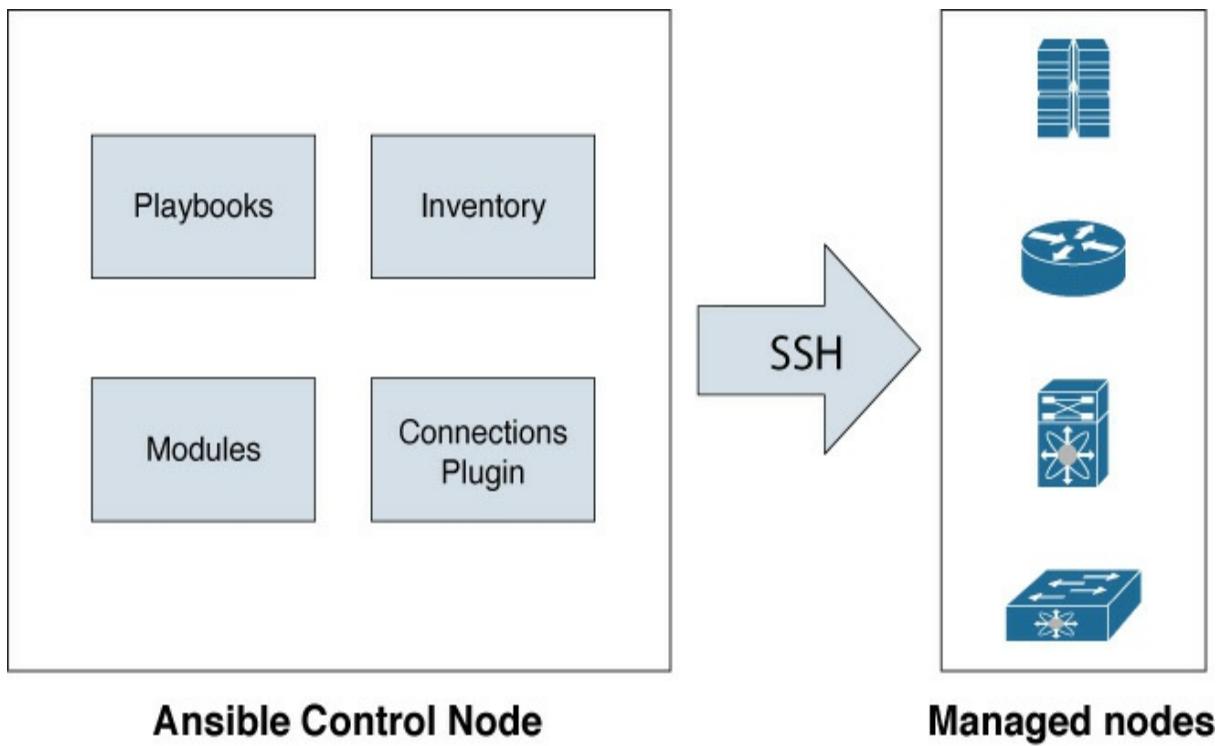


Figure 4-1 Ansible Architecture

Ansible uses YAML as its data format. Recall from [Chapter 2, “Data for Network Automation,”](#) that YAML is a human-friendly format.

Ansible has a modular architecture, which makes it an extensible tool. By default, Ansible comes with a set of modules. A *module* is a script written in a programming language (typically Python) that Ansible uses to interact with the nodes. You can download new modules or create your own. Most of the time, the available modules cover common operations, such as configuring devices, retrieving information from network devices, or interacting with specific software. However, when you can't find a module that suits your need, you can go to Ansible Galaxy, which is a repository where you can find many different modules. Ansible has a huge community behind it, providing supporting.

Module operations are *idempotent*. This means that operations are executed only once, if the an operation is repeated more than one time, Ansible will not do any modification and output to the user a not changed message. Ansible understands whether any modification is needed and executes only the needed modification. For example, if you instruct Ansible to install

Apache on a server that currently doesn't have Apache installed, the first time it executes that playbook, it installs Apache. If you rerun the same playbook, it will not make any modification because Apache is already installed.

Note

If you build your own modules, it is possible that they will not exhibit idempotent properties. This logic must be programmed into a module.

Installing Ansible

Before you can install Ansible, you must have Python installed. You only need to install Ansible in your control node.

Ansible can be installed on most operating systems, including Red Hat, Debian, CentOS, and macOS. However, it cannot be installed on Windows.

When you install Ansible on a control node, it is important to consider the proximity of this node to the managed nodes as the distance can be a critical factor for latency. If you plan to manage a cloud environment, having a control node in the cloud is a good idea.

There are several Ansible versions, and it is recommended to install the latest version recommended unless you have a specific requirement that must be addressed with a particular version.

Because Ansible is a Python package, it can be installed using **pip**, which is a Python package manager. You can also install Ansible with your operating system package's manager if you have one (for example, **yum** on CentOS).

You need to follow two required steps to install Ansible in a macOS workstation:

Step 1. Download and install **pip**.

Step 2. Install Ansible.

[Example 4-1](#) shows the command syntax of these two steps. The two first

commands represent step 1, and the third command represents step 2.

Note

macOS computers ship with Python installed. Depending on your operating system, you might have to install Python as a first step.

Example 4-1 *Installing Ansible on macOS*

```
$ curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
$ python get-pip.py --user
$ sudo python -m pip install ansible
```

After the installation finishes, you can verify your version with the command shown in [Example 4-2](#).

Example 4-2 *Verifying the Ansible Version*

```
$ ansible --version
ansible 2.9.13
  config file = None
  configured module search path = ['/Users/ivpinto/.ansible/plugins',
 '/usr/share/ansible/plugins/modules']
  ansible python module location =
/usr/local/Cellar/ansible/2.9.13_1/libexec/lib/python3.9/site-packages
  executable location = /usr/local/bin/ansible
  python version = 3.9.0 (default, Oct 29 2020, 13:07:11) [Clang 1200.0.32.21]
```



Note

For avid macOS users, installing Ansible with Homebrew is also possible.

Inventory

An inventory file, sometimes referred to as a *hosts* file, is where you define which nodes are your target hosts (that is, managed nodes). When you run a playbook, you need to run it against specific machines, which you define in the inventory file.

By default, Ansible tries to find hosts in the path /etc/ansible/hosts, but it is a best practice to create specific inventory files and refer to them when you execute playbooks.

An inventory file can have one of several formats: YAML, JSON, or INI. In this chapter, we focus on YAML.

A YAML inventory should start with the *all* group, which can have hosts, variables, and children.

Hosts are the targets. They can be represented with hostnames or IP addresses. [Example 4-3](#) shows a valid inventory file.

Example 4-3 Ansible Inventory

```
$ cat hosts.yml
---
all:
  hosts:
    10.10.10.1:
    10.10.10.2:
    10.10.10.3:
    switch01.example.com:
```

Variables are characteristics of hosts. They can be applied to single hosts or to host groups. In [Example 4-4](#), host 10.10.10.1 has the variable *status* with the value *prod*. Furthermore, all hosts have the variable *country* with the value *Portugal* assigned.

Example 4-4 Ansible Inventory with Variable Assignments

```
$ cat hosts.yml
```

```

---
all:
  hosts:
    10.10.10.1:
      status: "prod" #THIS IS A HOST VARIABLE
    10.10.10.2:
    10.10.10.3:
      switch01.example.com:
vars:
  country: "Portugal" #THIS IS A GROUP VARIABLE

```

Children are child groups. They can have their own hosts, variables, and children. Grouping is very useful. You should group hosts that have the same characteristics (for example, operating system, physical location). Grouping is helpful when you execute your playbooks.

[Example 4-5](#) shows the creation of two groups (Portugal and Spain) that identify in the countries where devices are geographically located. When you execute this playbook, by using these groups, you can make changes to devices in a specific country by using only the appropriate group keyword.

Example 4-5 Ansible Inventory with Groupings

```

$ cat hosts.yml
---
all:
  children:
    Portugal: #THIS IS A GROUP
      hosts:
        10.10.10.1:
          status: "prod"
        switch01.example.com:
    Spain: #THIS IS ANOTHER GROUP
      hosts:
        10.20.20.1:

```

There are two default groups in this example: *all* and *ungrouped*. *all* contains

all hosts, and *ungrouped* is for hosts that are not part of a group besides the group *all*. These are implicit concepts, and they may be omitted from the inventory file. However, as a general rule, an explicit definition is better than implicit one.

[Example 4-6](#) helps you understand the concept of groups. In this example, the host 10.30.30.1 is not part of any group besides the group *all*, so it is part of *ungrouped*.

Example 4-6 Ansible Inventory with Ungrouped Hosts

```
$ cat hosts.yml
---
all:
  children:
    Portugal:
      hosts:
        10.10.10.1:
          status: "prod"
        switch01.example.com:
    Spain:
      hosts:
        10.20.20.1:
hosts:
  10.30.30.1:
```

You can verify your inventory with an Ansible command. All you need is to have your inventory in a file format and execute the command **ansible-inventory** on it.

If you save the inventory in [Example 4-6](#) in a file named hosts.yml, you can execute **ansible-inventory** and get the output shown in [Example 4-7](#).

Example 4-7 Verifying an Inventory as Ansible Sees It

```
$ ansible-inventory -i hosts.yml --list
{
  "Portugal": {
```

```

    "hosts": [
        "10.10.10.1",
        "switch01.example.com"
    ]
},
"Spain": {
    "hosts": [
        "10.20.20.1"
    ]
},
"_meta": {
    "hostvars": {
        "10.10.10.1": {
            "status": "prod"
        }
    }
},
"all": {
    "children": [
        "Portugal",
        "Spain",
        "ungrouped"
    ]
},
"ungrouped": {
    "hosts": [
        "10.30.30.1"
    ]
}
}

```

As previously mentioned, the groups *all* and *ungrouped* are implicit definitions. The inventory shown in [Example 4-8](#) displays exactly the same result, even though it omits the initial group *all*, if verified with the command **ansible-inventory**, as shown in [Example 4-7](#). Try it for yourself by copying the inventory to a file and executing the **ansible-inventory** command. Doing so allows you to see in action the concept of implicit definitions.

Example 4-8 Ansible Inventory: Implicit Definitions

```
$ cat hosts.yml
---
Portugal:
  hosts:
    10.10.10.1:
      status: "prod"
    switch01.example.com:
Spain:
  hosts:
    10.20.20.1
10.30.30.1:
```

Tip

Run the **inventory** command for all of the previous inventory examples, and you can see how Ansible parses them, which helps you become familiar with the syntax.

In the event that your environment consists of a large number of sequential hosts, you can use a range function to define them, as shown here:

```
---
all:
  children:
    Portugal:
      hosts:
        10.10.10.[1:3]: #This represents 10.10.10.1, 10.10.10.2, and
```

This function also supports alphanumeric ranges.

Maintaining an inventory file is fine if your infrastructure does not regularly experience many changes or does not span thousands of devices. In case it does, however, Ansible offers you the possibility of having a dynamic inventory.

Dynamic inventories are scripts that collect information about your machines and return it to Ansible in JSON format. They can collect this information from multiple sources (for example, databases, files) using several techniques (for example, APIs, SSH).

This is a fairly advanced topic, but there are many prebuilt scripts for common use cases (for example, Amazon Web Services, OpenStack, Microsoft Azure) that you may be able to use.

Variables

Just as in other tools and programming languages, variables in Ansible are variable fields that can have values assigned. You can use them to parametrize and reuse the same value.

In Ansible you can assign variables by using the inventory, playbooks, files, or at runtime.

You saw in the previous section that you can assign variables to hosts or groups by using the inventory. In that case, you saw how to use the inventory file directly. However, Ansible also searches for directories named *host_vars* and *group_vars* in the inventory directory. Inside those directories, you can create files with the same name as your hosts or groups. Using this structure is the preferred way to assign variables to hosts. Example 4.9 shows this structure.

Example 4-9 Assigning Variables Using a Folder Structure

```
$ tree
.
├── group_vars/
│   └── Portugal.yaml
├── host_vars/
│   └── 10.10.10.1.yaml
└── hosts.yaml

$ cat hosts.yaml
---
```

```
all:
  children:
    Portugal:
      hosts:
        10.10.10.[1:2]:
      hosts:
        switch01.example.com:

$ cat group_vars/Portugal.yaml
---
type: "CAT9k"

$ cat host_vars/10.10.10.1.yaml
---
status: "prod"

$ ansible-inventory -i hosts.yaml --list
{
  "Portugal": {
    "hosts": [
      "10.10.10.1",
      "10.10.10.2"
    ]
  },
  "_meta": {
    "hostvars": {
      "10.10.10.1": {
        "status": "prod",
        "type": "CAT9k"
      },
      "10.10.10.2": {
        "type": "CAT9k"
      }
    }
  },
  "all": {
    "children": [
      "Portugal",
      "ungrouped"
    ]
  }
}
```

```
        ]
    },
    "ungrouped": {
        "hosts": [
            "switch01.example.com"
        ]
    }
}
```

Note

The **tree** command is not natively installed in most operating systems. If you want to use it, you must install it.

In [Example 4-9](#), you can see that all hosts from the group *Portugal* have the type *CAT9K*. But only the host *10.10.10.1* has the status *prod*. This is the behavior reflected in the files under the *group_vars* and *host_vars* directories.

You can also define variables in playbooks. You define them at the top of a playbook file by using the keyword *vars*, as shown in this example:

```
- name: Ansible Playbook to change TACACS keys for ASA Devices
  hosts: asa
  vars:
    tacacs_key: "C!sc0123"
```

Yet another way definite define variables is by using files. You can create YAML files with key/value pairs for your variables and refer to them in your playbooks.

[Example 4-10](#) shows how to create a new file called *variables.yaml* and define two key/value pairs (*tacacs_key* and *tacacs_servers*). In this playbook, *playbook.yaml*, you can refer to this file in the *var_files* keyword by specifying the file path to the previously created variables file.

Example 4-10 Assigning Variables Using Variable Files

```
$ cat variables.yaml
```

```
---
tacacs_key: "C!sc0123"
tacacs_server: "10.10.10.253"

$ cat playbook.yaml
---
- name: Ansible Playbook to change TACACS keys for ASA Devices
  hosts: asa
  vars_files:
    - variables.yaml
```

As mentioned earlier, you can specify variables at runtime. You achieve this by using the `-e` argument when running a playbook, as shown here:

```
$ ansible-playbook playbook.yaml -e "tacacs_key=C!sc0123"
```

You will learn more on running playbooks later in this chapter.

You can pass a file reference instead of a key/value pair by using the `@` symbol as a prefix to the file path, as shown here:

```
$ ansible-playbook playbook.yaml -e "@variables.yaml"
```

Separating the variables from the playbooks allows you to reuse the same playbooks in different environments. However, there are some variables that are tied to host types (for example, `ansible_network_os`). They can be defined in the inventory.

It is a good idea to avoid defining the same variable in different places. If you are working as part of a team, the team members should agree where the variables will be defined.

Tip

Separating variables from playbooks is crucial when you want to store them in a source control repository.

Besides the variables you have already seen, there is another type of variable, referred to as *registered variable*. This type is defined as a result of tasks, as

shown here:

```
---
- hosts: all
  tasks:
    - shell: time
      register: result
```

After a variable is defined independently of the type, you can refer to it by using the following syntax:

```
{{ variable_name }}
```

Example 4-11 shows a playbook (play.yml) that defines a variable (*example*) and prints its value. This playbook is executed on the localhost.

Example 4-11 Ansible Playbook Using a Defined Variable

```
$ cat play.yml
---
- hosts: all
  vars:
    example: "This is an example variable"
  tasks:
    - debug:
        msg: '{{ example }}'

$ ansible-playbook -c local -i "localhost," play.yml

PLAY [all] ****
TASK [Gathering Facts]
*****
ok: [localhost]

TASK [debug]
*****
ok: [localhost] => {
    "msg": "This is an example variable"
}
```

```
PLAY RECAP ****
localhost : ok=2     changed=0     unreachable=0
ignored=0
```

Some variables should be kept secret, such as for credentials. An Ansible Vault allows you to store variables in encrypted format. You should use it instead of saving your secrets in plaintext.

[Example 4-12](#) shows how to create an Ansible Vault and use it in a playbook. It starts by creating the Vault with a password and writing secrets inside it—in this case a key/value pair (the key *example* and the value *This is an example variable*).

The example then verifies that the Vault was saved correctly using the password. If you try to access the file—by using **cat**, for example—you see only gibberish.

The playbook in [Example 4-12](#) is just like the one in [Example 4-11](#), but it imports the Vault file. Notice that it refers to the variable by using the same syntax.

When you execute the playbook in this case, you get exactly the same output as in [Example 4-11](#). The advantage here is that the secret information is kept secret, protected by a password.

Example 4-12 Ansible Playbook Using a Variable in a Vault

```
$ ansible-vault create vault.yml
New Vault password:
Confirm New Vault password:

Write your secret information inside:
example: "This is an example variable"

$ ansible-vault view vault.yml
Vault password:
example: "This is an example variable"
```

```
$ cat vault.yml
$ANSIBLE_VAULT;1.1;AES256
34616130343962613037383861653834333646666633064343163323231366264
666336313135323735346333361663762313037336437330a3034646137653562
326234346433366535643963386439616362613934643861373136646462393064
3236633431633439320a363532626361616632363833376665653934653333262
343838393536346463626238353364346335303536313962643165396565306135
613965343435636464613539346134323239356262636233

$ cat play.yaml
- hosts: all
  vars_files:
    - ./vault.yml
  tasks:
    - debug:
        msg: '{{ example }}'

$ ansible-playbook -c local -i "localhost," play.yml --ask-vault-p
Vault password:

PLAY [all] ****
TASK [Gathering Facts]
*****
ok: [localhost]

TASK [debug]
*****
ok: [localhost] => {
    "msg": "This is an example variable"
}

PLAY RECAP ****
localhost                  : ok=2      changed=0      unreachable=0
rescued=0      ignored=0
< >
```

Playbooks

You have seen several Ansible playbooks already in this book. An Ansible *playbook* is composed by a number of *plays*, which are, in turn, composed of a number of *tasks*, which define actions. [Figure 4-2](#) provides a visual representation of this structure.

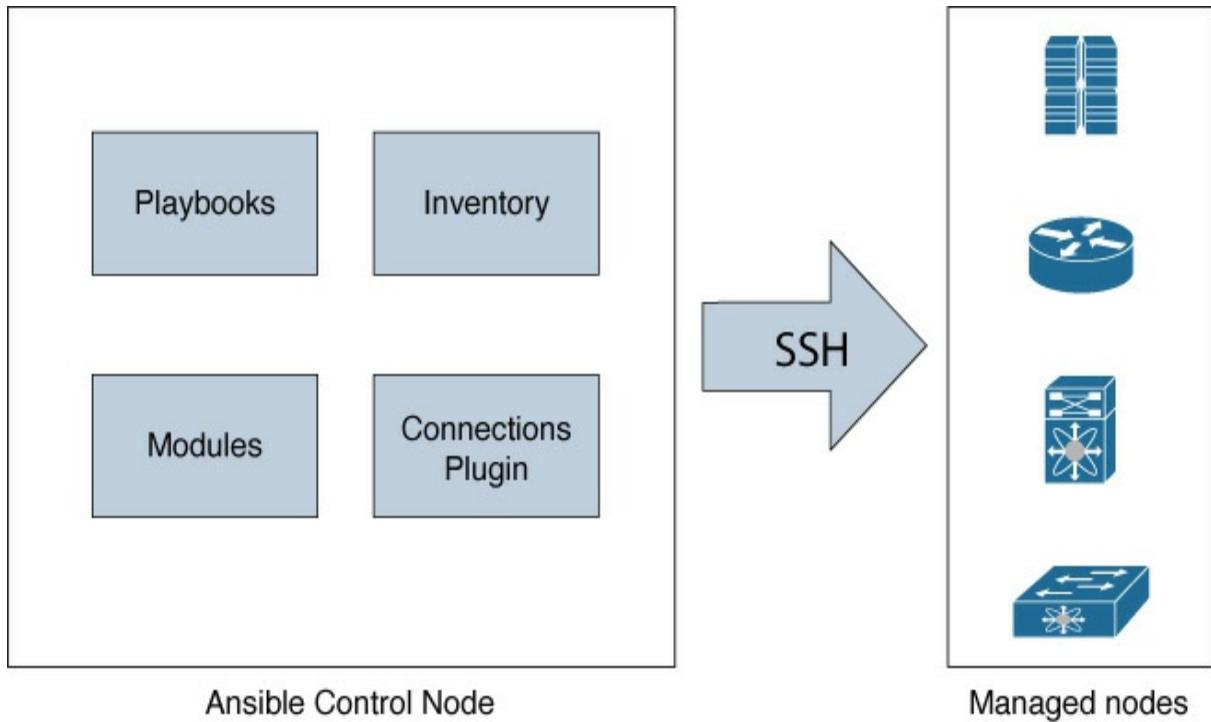


Figure 4-2 *The Structure of Ansible Playbooks*

Playbooks are written in YAML. A simple playbook could contain a single play, and a complicated one could achieve a complex workflow. In essence, a playbook is a list of instructions that you execute on remote systems (managed nodes).

Playbooks use modules, which are programs written in any language (but typically Python) that implement the logic needed to achieve a particular task.

[Example 4-13](#) shows a task named *Configure NTP* that uses the module *ios_ntp*. This module implements the logic needed to push the configurations you provide to the network device (for example, connection type, translation from Ansible inputs to CLI commands).

Example 4-13 Ansible Playbook to configure NTP using `ios_ntp`

```
$ cat play.yaml
---
- name: PLAY - Update NTP Configurations
  hosts: all
  tasks:
    - name: Configure NTP
      cisco.ios.ios_ntp:
        server: 10.0.100.1
        source_int: Loopback0
        logging: false
        state: present
```

You may have noticed the `hosts` keyword at the top of the play. A playbook needs to be instructed on what to run against, and you provide this instruction by using the `hosts` keyword. This is where you can refer to the `hosts` groups you created in your inventory file.

[Example 4-14](#) shows two host groups defined, `ios` and `nxos`, as well as the `all` group. If you executed `playbook1.yaml`, it would run the tasks in all four switches; `playbook2.yaml` would only run tasks on Switches 3 and 4.

Example 4-14 Running a Playbook for a Specific Group of Hosts

```
$ cat inventory.yaml
all:
  children:
    ios:
      hosts:
        switch_1:
          ansible_host: "10.10.10.1"
        switch_2:
          ansible_host: "10.10.10.2"
    nxos:
      hosts:
        switch_3:
```

```
ansible_host: "10.10.10.3"
switch_4:
    ansible_host: "10.10.10.4"

$ cat playbook1.yaml
---
- name: PLAY - Update NTP Configurations
  hosts: all

$ cat playbook2.yaml
---
- name: PLAY - Update NTP Configurations
  hosts: ios
```

The following sections explore some of the logical constructs that you can use to build playbooks: conditionals, blocks, and loops.

Conditionals

You use conditionals when you do not want a task to execute always but you want to execute some tasks based on a condition being true. In Ansible, you achieve this by using the *when* statement. Ansible evaluates the *when* statement before running the task. You can use this feature based on many different conditions, some of the most common of which are registered variables and *ansible_facts*.

Typically, the *when* statement is used based on predefined variables, or facts gathered from devices.

Note

Ansible automatically gathers facts about the target host. You can access this data in the *ansible_facts* variable.

[Example 4-15](#) shows a playbook with two tasks: The first one prints the hostname of the device, and the second one configures VLAN 656 if the

device hostname is switch01. You can see from the execution output that the hostname of the switch is POD4-DC-01 and not switch01. Therefore, Ansible skips the VLAN configuration task.

It is important to note that this example and the following ones require a switch to interact with, and you must build your own inventory file as inventory.yml in order to be able to run the playbook.

Example 4-15 Configuring a VLAN Only on a Switch with a Specific Hostname

```
$ cat vlans.yaml
---
- name: Ansible Playbook to configure vlans for IOS Devices
  hosts: ios

  tasks:
    - name: print facts
      debug:
        msg: "{{ ansible_net_hostname }}"

    - name: Configure vlan 656
      cisco.ios.ios_vlans:
        config:
          - name: test_vlan
            vlan_id: 656
            state: active
            shutdown: disabled
      when: ansible_net_hostname == "switch01"

$ ansible-playbook vlans.yaml -i inventory.yml

PLAY [Ansible Playbook to configure vlans for IOS Devices] ****
*****
TASK [Gathering Facts]
*****
ok: [switch_1]

TASK [print facts]
```

```

*****
ok: [switch_1] => {
    "msg": "POD4-DC-01"
}

TASK [Configure vlan 656]
*****
skipping: [switch_1]

PLAY RECAP
*****
switch_1 : ok=2     changed=0     unreachable=0
rescued=0   ignored=0
< >

```

An important aspect of conditionals is that they can be applied to blocks of tasks. Ansible uses blocks to group tasks.

All tasks inside a block are applied only if the condition evaluates to true. [Example 4-16](#) shows an enhancement of the playbook from [Example 4-15](#). The first task continues to simply print the device's hostname. However, the second task is now a block that entails two tasks: It configures VLAN 656 and VLAN 668. Nonetheless, the condition remains the same: The hostname must be switch01. From the execution of the playbook in [Example 4-16](#), you can see that because the device's hostname still does not match the condition, both VLAN configuration tasks are skipped.

Example 4-16 Configuring Two VLANs Only on a Switch with a Specific Hostname

```

$ cat vlans.yaml
---
- name: Ansible Playbook to configure vlans for IOS Devices
  hosts: ios

  tasks:
    - name: print facts
      debug:

```

```
msg: "{{ ansible_net_hostname }}"

- name: block of vlan tasks
  block:
    - name: Configure vlan 656
      cisco.ios.ios_vlans:
        config:
          - name: test_vlan
            vlan_id: 656
            state: active
            shutdown: disabled
    - name: Configure vlan 668
      cisco.ios.ios_vlans:
        config:
          - name: test_vlan
            vlan_id: 668
            state: active
            shutdown: disabled
  when: ansible_net_hostname == "switch01"
```

```
$ ansible-playbook vlans.yaml -i inventory.yml
```

```
PLAY [Ansible Playbook to configure vlans for IOS Devices]*****  
  
TASK [Gathering Facts]  
*****  
ok: [switch_1]  
  
TASK [print facts]  
*****  
ok: [switch_1] => {  
  "msg": "POD4-DC-01"  
}  
  
TASK [Configure vlan 656]  
*****  
skipping: [switch_1]  
  
TASK [Configure vlan 668]
```

```
*****
skipping: [switch_1]

PLAY RECAP
*****
switch_1 : ok=2    changed=0    unreachable=0
< >
```

Tip

In Ansible, blocks are typically used for error handling.

Loops

A loop involves iterating over something. You can use a loop when you want to repeat the same task multiple times or with different inputs. In Ansible, you can achieve this with the keywords *loop* and *item*.

The playbook shown in [Example 4-17](#) is a modification of the one in [Example 4-16](#). This version uses a loop on a task to configure two different VLANs instead of two different tasks. Another difference between the playbooks is that the conditional block has been removed in this case.

From the execution of the playbook in this example, you can see both VLANs being configured on the switch.

Example 4-17 Configuring Two VLANs on a Switch Using a Loop

```
$ cat vlans.yaml
---
- name: Ansible Playbook to configure vlans for IOS Devices
  hosts: ios

  tasks:
    - name: print facts
      debug:
```

```

msg: "{{ ansible_net_hostname }}"

- name: Configure vlans
  cisco.ios.ios_vlans:
    config:
      - name: test_vlan
        vlan_id: "{{ item }}"
        state: active
        shutdown: disabled
    loop:
      - 656
      - 668

$ ansible-playbook vlans.yaml -i inventory.yml

PLAY [Ansible Playbook to configure vlans for IOS Devices] *****

TASK [Gathering Facts] *****
ok: [switch_1]

TASK [print facts] *****
ok: [switch_1] => {
    "msg": "POD4-DC-01"
}

TASK [Configure vlans] *****
ok: [switch_1] => (item=656)
ok: [switch_1] => (item=668)

PLAY RECAP *****
switch_1 : ok=3     changed=0     unreachable=0
rescued=0   ignored=0

```

You can use the same technique shown in [Example 4-17](#) to iterate over lists of dictionaries. This is a useful feature when you have more than one attribute to iterate over.

[Example 4-18](#) shows a playbook that configures two VLANs with specific

names. It defines the VLAN IDs and names using a dictionary format and loops over them in the configuration task.

Example 4-18 Configuring Two Named VLANs on a Switch by Using a Loop

```
$ cat vlans.yaml
---
- name: Ansible Playbook to configure vlans for IOS Devices
  hosts: ios

  tasks:
    - name: print facts
      debug:
        msg: "{{ ansible_net_hostname }}"

    - name: Configure vlans
      cisco.ios.ios_vlans:
        config:
          - name: "{{ item.name }}"
            vlan_id: "{{ item.vlan_id }}"
            state: active
            shutdown: disabled
        loop:
          - { name: 'test_vlan1', vlan_id: '656' }
          - { name: 'test_vlan2', vlan_id: '668' }

$ ansible-playbook vlans.yaml -i inventory.yml

PLAY [Ansible Playbook to configure vlans for IOS Devices]
*****
TASK [Gathering Facts]
*****
ok: [switch_1]

TASK [print facts]
*****
ok: [switch_1] => {
```

```

        "msg": "POD4-DC-01"
    }

TASK [Configure vlans]
*****
ok: [switch_1] => (item={'name': 'test_vlan1', 'vlan_id': '656'})
ok: [switch_1] => (item={'name': 'test_vlan2', 'vlan_id': '668'})

PLAY RECAP
*****
_1                               : ok=3      changed=0      unreachable=0      failed=0
rescued=0      ignored=0

<   [REDACTED]   >

```

There is another keyword, *with_items*, that you might encounter in place of *loop*. *loop* is a newer version that replaces *with_items* in most scenarios in newer versions of Ansible. You might see *with_items*, however, and you should know that it has the same structure as *loop*.

Tip

In older playbooks, you will more often find *with_items* than *loop*. You may choose to convert *with_items* to *loop* to keep all your playbooks consistent.

[Example 4-19](#) shows the playbook from [Example 4-17](#) adapted to use the keyword *with_items*.

Example 4-19 A Playbook to Configure Two VLANs on a Switch by Using *with_items*

```

$ cat vlans.yaml
---
- name: Ansible Playbook to configure vlans for IOS Devices
  hosts: ios

  tasks:

```

```

- name: print facts
  debug:
    msg: "{{ ansible_net_hostname }}"

- name: Configure vlans
  cisco.ios.ios_vlans:
    config:
      - name: test_vlan
        vlan_id: "{{ item }}"
        state: active
        shutdown: disabled
    with_items:
      - 656
      - 668

```

Note

The Ansible keyword is actually `with_<word>`, where `<word>` can be replaced with a value other than `items` that you might want to use in specific scenarios. For example, you might want to use the keyword `with_indexed_items`.

There is a special kind of loop that is worth highlighting: It uses the keyword `until`. This keyword retries a task until a certain condition is met, which is particularly useful for verifications. The syntax is as follows:

```

- name: Check on an async task
  async_status:
    job_id: "{{ vlan_status.ansible_job_id }}"
  register: job_result
  until: job_result.finished
  retries: 5
  delay: 10

```

The keyword `until` is used with the condition that must be met, and it causes the playbook to retry to repeat the task a specified number of times. `delay` indicates the number of seconds that it waits between attempts.

Handlers

A handler is a special kind of task in a playbook. It behaves like a task but is executed only if notified to run. Tasks can notify handlers to run. Why would you need handlers? There are scenarios in which you just want something to be done if a task has run. For example, you might want to save the running configuration to the startup configuration only if modifications have been made to the running configuration.

By default, handlers only run at the end of an Ansible play. This default behavior means that handlers run only once even if notified by several tasks. This can be useful if you have several tasks making configuration changes but you only really want to save the configuration to the startup configuration after all changes have been made.

Note

You can execute handlers before the end of a play by using Ansible metadata.

The playbook in [Example 4-20](#) tries to configures VLAN 656 on a switch. As you can see, if it is already configured, nothing happens. However, if it is not configured, the Ansible playbook creates this VLAN and notifies a handler that saves the switch's configuration.

Example 4-20 A VLAN Configuration Playbook Using Handlers to Save the Configuration

```
$ cat handlers.yml
---
- name: Ansible Playbook to configure vlans for IOS Devices
  hosts: ios
  gather_facts: no

  tasks:
    - name: Configure vlan 656
```

```

cisco.ios.ios_vlans:
  config:
    - name: test_vlan
      vlan_id: 656
      state: active
      shutdown: disabled
    notify: Save configuration

handlers:
  - name: Save configuration
    cisco.ios.ios_config:
      save_when: modified

```

You can see the different execution steps when the VLAN exists and when it does not from the Ansible's output on [Example 4-21](#). In the first execution, the VLAN still did not exist, and so the task was run, and the handler was triggered, saving the configuration. In the second execution, the VLAN already existed, and Ansible simply informed you that everything was as intended.

Example 4-21 Possible Differences in Execution of the Playbook in [Example 4-20](#)

```

$ ansible-playbook handlers.yml -i inventory.yml

PLAY [Ansible Playbook to configure vlans for IOS Devices] *****

TASK [Configure vlan 656]
*****
changed: [switch_1]

RUNNING HANDLER [Save configuration]
*****
changed: [switch_1]

PLAY RECAP *****
switch_1 : ok=2     changed=2     unreachable=0
rescued=0   ignored=0

```

```

$ ansible-playbook handlers.yml -i inventory.yml

PLAY [Ansible Playbook to configure vlans for IOS Devices]
*****
TASK [Configure vlan 656]
*****
ok: [switch_1]

PLAY RECAP *****
switch_1 : ok=1    changed=0    unreachable=0
rescued=0  ignored=0

```

Another important thing to note about handlers is that you can trigger multiple handlers from a single task. This is useful when you want to execute multiple operations. You define a list of handler names for the *notify* value.

In [Example 4-22](#), the task that configures VLAN 656 notifies two handlers: *Save configuration* and *Copy configuration*. This is a common operation in networking: When you perform a configuration update, you typically want to commit that configuration into the device's long-term memory and also export it for safekeeping.

Example 4-22 An Ansible Task with Multiple Handlers

```

$ cat handlers.yml
---
- name: Ansible Playbook to configure vlans for IOS Devices
  hosts: ios
  gather_facts: yes

  tasks:
    - name: Configure vlan 656
      cisco.ios.ios_vlans:
        config:
          - name: test_vlan

```

```

    vlan_id: 656
    state: active
    shutdown: disabled
    notify:
      - Save configuration
      - Copy configuration

handlers:
  - name: Save configuration
    cisco.ios.ios_config:
      save_when: modified
  - name: Copy configuration
    copy:
      content: "{{ ansible_facts.net_config }}"
      dest: "router_bkup.txt"

```

Having multiple handlers can be useful, as shown here. However, when you have multiple handlers, your playbooks become increasingly complex and hard to read. An easier way to notify all your handlers is to use topics instead of enumerating all their names under *notify*.

Example 4-23 re-creates the [Example 4-22](#), in which a VLAN configuration task notifies two different handlers (*Save configuration* and *Copy configuration*). This example creates the topic *configuration changes*, and both handlers subscribe to it. During the play’s execution, the configuration task publishes to this topic instead of calling the handlers by name.

Example 4-23 Ansible Task with Multiple listener Handlers

```

$ cat handlers.yml
---
- name: Ansible Playbook to configure vlans for IOS Devices
  hosts: ios
  gather_facts: yes

  tasks:
    - name: Configure vlan 656
      cisco.ios.ios_vlans:

```

```

config:
  - name: test_vlan
    vlan_id: 656
    state: active
    shutdown: disabled
    notify: "configuration changes"

handlers:
  - name: Save configuration
    cisco.ios.ios_config:
      save_when: modified
    listen: "configuration changes"
  - name: Copy configuration
    copy:
      content: "{{ ansible_facts.net_config }}"
      dest: "router_bkup.txt"
    listen: "configuration changes"

```

You can see that the code becomes more readable and decouples the handlers from their names. This example shows only a single task and two handlers, but the larger the scale (for example, tens of tasks or tens of handlers), the more this technique becomes a time and complexity saver.

Executing a Playbook

At this point, you have seen several playbooks executed. You execute a playbook from a control node by using the command **ansible-playbook**. This command can take a number of options, the most common of which are listed in [Table 4-1](#).

Table 4-1 *ansible-playbook* Command Options

Option	Description
<code>-i</code>	Specifies the host's file path or a comma-separated host list.
<code>-e</code>	Specifies variables as key/value pairs or YAML/JSON lists.
<code>-vvv</code>	Enables more verbose output, which is useful for troubleshooting
<code>-C</code>	Makes no changes on the target host and tries to predict what will change. Also known as check mode. This is not supported by all modules.
<code>-u</code>	Specifies the user to use for the connection.
<code>--ssh-common-args</code>	Specifies the SSH options for the connection.
<code>-f</code>	Specifies the number of task forks.

Tip

In a network, it is typical for managed systems to be behind a bastion/jump host. You can use `--ssh-common-args` with **ProxyCommand** to achieve this connectivity.

To execute a playbook, you use the following syntax:

```
$ ansible-playbook [options] playbook_name.yaml
```

Another option for running Ansible tasks is to use ad hoc commands. Ad hoc commands give you an easy way of executing a task quickly, but they are not reusable, as playbooks are. Ad hoc commands are often useful for doing quick tests or changes. The syntax for executing an ad hoc command is similar to the syntax for executing a playbook:

```
$ ansible host-pattern -m module [-a 'module arguments'] [-i inventory]
```

Note

In general, you want to use playbooks rather than ad hoc commands.

An example of usage of an ad hoc action could be creating a VLAN on a set of hosts:

```
$ ansible all -m cisco.ios.ios_vlan -a 'vlan_id=700' -i inventory.yml
```

This command executes the module `ios_vlan` on all hosts defined in `inventory.yml` with VLAN ID 700. The result is the creation of this VLAN in the event that it is not already present on the device. The following command execution is from an inventory file with a single switch (`switch_1`) where the VLAN 700 was not yet present:

```
$ ansible all -m cisco.ios.ios_vlan -a 'vlan_id=700' -i inventory.yml
switch_1 | CHANGED => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": true,
    "commands": [
        "vlan 700"
    ]
}
```

When you execute Ansible playbooks or ad hoc commands, behind the scenes, Ansible copies Python scripts (which are responsible for making your tasks work) to the managed node and executes them there; this is the traditional workflow. This means the Python code responsible for achieving a task is not run from the control node; rather, it is copied over to the managed node and executed there.

Some systems (for example, network devices) do not support Python, but Ansible still works on them. In these special cases, Ansible executes the Python scripts locally on the control node. This is typically the case for connections that use APIs or, as previously mentioned, network devices.

By default, tasks are run in five hosts in parallel, but you can change this by using the `-f` parameter on the command line. The ability to execute a large number of hosts in parallel depends on the control node capacity.

Ansible uses a linear strategy so that all hosts in your inventory run each task

before any other host can start the next task. You can change this to have Ansible execute tasks freely without waiting for other hosts or to execute all tasks on a host before moving to the next host. These changes should be applied only if necessary.

[Example 4-24](#) shows how a serial execution would look. This example makes use of an inventory file with four different devices as part of a group named *ios*. To run this example, you must create your own inventory file with devices that are reachable from your workstation and part of a device group called *ios*.

Example 4-24 Ansible Serial Execution of Tasks

```
$ cat serial.yml
---
- name: Serial playbook
  hosts: ios
  serial: 2
  gather_facts: False

  tasks:
    - name: Get NTP status 1
      cisco.ios.ios_command:
        commands: show ntp status
    - name: Get NTP status 2
      cisco.ios.ios_command:
        commands: show ntp status

$ ansible-playbook serial.yml -i inventory.yml

PLAY [Serial playbook]
*****
TASK [Get NTP status 1]
*****
changed: [ios1]
changed: [ios2]

TASK [Get NTP status 2]
```

```

*****
changed: [ios1]
changed: [ios2]

PLAY [ios]
*****

TASK [Get NTP status 1]
*****
changed: [ios3]
changed: [ios4]

TASK [Get NTP status 2]
*****
changed: [ios3]
changed: [ios4]

```



As you can see in [Example 4-24](#), both tasks (NTP status 1 and 2) are executed on hosts *ios1* and *ios2* before being started on hosts *ios3* and *ios4*. This could be particularly useful for rolling updates.

When executing a playbook, you might want some tasks to be executed only on the first host of your inventory. For example, you might know that the result will be the same for all the other hosts, and it would be a waste of CPU cycles and network bandwidth to rerun a playbook on all the others. You can achieve this by using the *run_once* flag, as shown in [Example 4-25](#). Here, regardless of the number of hosts in your inventory, you would only retrieve the time from the first one because all the clocks in a network environment are typically synchronized to the same NTP server.

Example 4-25 Ansible *run_once* Task

```

$ cat run_once.yml
---
- name: run once playbook
  hosts: ios
  gather_facts: False

```

```
tasks:
  - name: Get NTP status 1
    cisco.ios.ios_command:
      commands: show clock
    run_once: true
```

Ansible tasks in a playbook execution run synchronously. This means Ansible holds the SSH connection open until a task completes, without progressing to the next task. Yet you might need a different behavior for some of your tasks, such as for long-running tasks, as in these use cases:

- For tasks timing out due to the SSH session timer
- To run tasks concurrently

For such use cases, you can configure asynchronous tasks. This works with both playbooks and ad hoc commands.

You use two keywords when configuring asynchronous tasks: *async* and *poll*. You use *async* to configure the maximum wait timer for an Ansible task. *poll* is the time interval that Ansible uses to check the status of the running task; by default, it is 10 seconds.

The Ansible task in [Example 4-26](#) sets the maximum timer (*async*) at 45 seconds and verifies its status every 5 seconds (*poll*).

Example 4-26 Asynchronous Ansible Task

```
$ cat async.yml
---
- name: Asynchronous playbook
  hosts: all
  gather_facts: False

  tasks:
    - name: Configure vlan 656
      cisco.ios.ios_vlans:
        config:
          - name: test_vlan
```

```
vlan_id: 656
state: active
shutdown: disabled
async: 45
poll: 5
```

[Example 4-26](#) shows how you achieve the first use case: long-running tasks. You simply modify these timers accordingly. However in this case, Ansible does not move to the next task but blocks further tasks until it either succeeds, fails, or goes over the *async* timer.

You can achieve concurrent tasks by defining a poll interval of 0. In this case, Ansible starts the next task immediately. The playbook finishes without checking the status of the task because it simply starts it.

You can insert the task code shown in [Example 4-27](#) in a playbook and execute it more than once against the same switch. If you do, you see that the result always displays the changed status, even though the VLAN already exists after the first execution. This behavior is different from what you are accustomed to seeing from the previous examples. The reason for this is that Ansible is not checking the task status at all; it is simply executing the task.

Example 4-27 Asynchronous Nonblocking Ansible Task

```
tasks:
  - name: Configure vlan 656
    cisco.ios.ios_vlans:
      config:
        - name: test_vlan
          vlan_id: 656
          state: active
          shutdown: disabled
    async: 45
    poll: 0
```

Not verifying the status is sometimes referred to as “fire and forget.” It involves more overhead because it might mean executing actions that are not needed. However, there is a way to have concurrent tasks in a playbook and

still check their status. You achieve this by registering the ID of a previously triggered action and verifying its status later on.

Example 4-28 registers the configuration task and, in the second task (*Check on an async task*), verifies every 10 seconds whether it has completed; it makes five attempts, using the *async_status* module. These are sequential tasks, but you can have any number of tasks in between the asynchronous task and its verification to achieve true concurrency.

Example 4-28 Asynchronous Ansible Task Status Verification

```
tasks:
  - name: Configure vlan 656
    cisco.ios.ios_vlans:
      config:
        - name: test_vlan
          vlan_id: 656
          state: active
          shutdown: disabled
      async: 45
      poll: 0
      register: vlan_status

  - name: Check on an async task
    async_status:
      jid: "{{ vlan_status.ansible_job_id }}"
      register: job_result
      until: job_result.finished
      retries: 5
      delay: 10
```

Roles

Ansible playbooks can become long and hard to read if they accomplish complex workflows that entail several tasks. Ansible roles try to address this issue. Ansible roles are used to collate together several tasks that have a

common purpose.

An Ansible role has a specific directory structure that is modular and easily reusable. A role contains several directories: defaults, vars, tasks, files, templates, meta, and handlers. Each directory must contain a main.yml file. You do not need to populate all the directories but only the relevant ones for your role. [Table 4-2](#) describes these directories.

Table 4-2 *Ansible Role Directories*

Directory	Description
tasks	Stores tasks that the role executes.
handlers	Stores Ansible handlers needed for the role.
tests	Used if there are automated tests for this role.
defaults	Defines default variables, which are the lowest priority in the case of multiple definitions.
vars	Stores variables used within a role. This is yet another place for variables.
files	Stores files needed for a role.
templates	Stores templates needed for a role. Templates are very similar to files except that templates can be modified.
meta	Defines metadata for the authorship of a role.

An easy way to create the directory structure is by using **ansible-galaxy**, but you can also do it manually. The syntax to achieve it automatically is as follows:

```
$ ansible-galaxy init new_role
- Role new_role was created successfully
```

[Example 4-29](#) shows the Ansible Galaxy role default folder structure.

Example 4-29 *Ansible Galaxy Role Default Folder Structure*

```
$ tree
.
└── new_role
```

```
└── README.md
└── defaults
    └── main.yml
└── files
└── handlers
    └── main.yml
└── meta
    └── main.yml
└── tasks
    └── main.yml
└── templates
└── tests
    ├── inventory
    └── test.yml
└── vars
    └── main.yml
```

After you have created a folder structure, you can add to the specific main.yml file content that is relevant from your role definition.

[Example 4-30](#) shows a role to create VLANs. In this example, you see only the tasks folder within the role directory because it is the only one you need as you are only defining tasks. More complex roles may need other folders.

Example 4-30 Creating a VLAN on a Switch by Using a Custom Ansible Role

```
$ tree
.
├── inventory.yml
└── roles
    └── vlans
        └── tasks
            └── main.yml
└── vlans.yaml

$ cat vlans.yaml
---
```

```

- hosts: ios
  vars:
    req_vlan : { name: 'test_vlan1', vlan_id: '656' }
  roles:
    - vlans

$ cat roles/vlans/task/main.yml
---
- name: print facts
  debug:
    msg: "{{ ansible_net_hostname }}"

- name: Configure vlans
  cisco.ios.ios_vlans:
    config:
      - name: "{{ req_vlan.name }}"
        vlan_id: "{{ req_vlan.vlan_id }}"
        state: active
        shutdown: disabled

$ ansible-playbook -i inventory.yml vlans.yaml
PLAY [ios]
*****
TASK [Gathering Facts]
*****
ok: [switch_1]

TASK [vlans : print facts]
*****
ok: [switch_1] => {
    "msg": "POD2-GB-01"
}

TASK [Configure vlans]
*****
changed: [switch_1]

PLAY RECAP
```

```
*****
switch_1 : ok=3     changed=1     unreachable=0
rescued=0   ignored=0
```

```
POD2-GB-01#show vlan
```

VLAN	Name	Status	Ports
656	test_vlan1	active	

You can see in [Example 4-30](#) that inside the playbook, you only import the role and define the variables for the VLAN. All the task logic (that is, how to achieve the task) is defined in main.yml, inside the role's task directory.

The playbook in [Example 4-30](#) is much simpler (with only five lines) than the one shown in [Example 4-15](#), although they achieve the same goal, configuring VLANs on a switch. You can clearly see the power of roles when it comes to a playbook's readability; roles offload complex and long task logic to a different domain.

Tip

There are thousands of pre-created roles in the Ansible Galaxy repository that you can use. Before implementing your own roles, verify whether a role you want already exists. Using a pre-created role will save you a lot of time and effort.

Summary

This chapter describes a number of Ansible concepts. It starts by showing how to install Ansible and its requirements. Then it shows how to build an Ansible inventory from scratch, describing different ways of grouping hosts, possible data formats, and best practices.

This chapter also covers playbooks and their components:

- Plays
- Tasks
- Modules
- Variables
- Conditionals
- Loops
- Handlers

This chapter looks at each of these components to help you understand how to use them in network automation scenarios. Furthermore, this chapter explains how playbooks are executed against different types of target devices and the available execution options.

Finally, this chapter touches on Ansible roles. Roles help you modularize your playbooks so they are easier to read and understand. Roles are also great for sharing what you have created.

[Chapter 5, “Using Ansible for Network Automation,”](#) shows how to use the concepts presented in this chapter to build playbooks that perform common networking tasks, such as modifying device configurations or gathering information from show commands.

Review Questions

You can find answers to these questions in [Appendix A, “Answers to Review Questions.”](#)

1. What programming language was Ansible written in?
 - a. Java
 - b. Python
 - c. Ruby
 - d. JavaScript

- 2.** Which of the following are valid data formats for representing an Ansible inventory? (Choose two.)
- a.** YAML
 - b.**INI
 - c.** XML
 - d.** Python
- 3.** In the following inventory file, which hosts are part of the *ungrouped* group?

```
---  
all:  
  children:  
    Portugal:  
      hosts:  
        10.10.10.1:  
          status: "prod"  
        switch01.example.com:  
    Spain:  
      hosts:  
        10.20.20.1:  
    hosts:  
      10.30.30.1:
```

- a.** 10.20.20.1
 - b.** 10.10.10.1
 - c.** switch01.example.com
 - d.** 10.30.30.1
- 4.** In the inventory file shown in question 3, what does *status: "prod"* represent?
- a.** Group variable

- b.** Host variable
 - c.** Child
 - d.** Host
- 5.** If you want to upload your playbooks to a shared repository, where should you typically define your variables?
- a.** In the host variables folder
 - b.** In a separate file that is uploaded to the repository
 - c.** In the playbook
 - d.** In the execution CLI
- 6.** When writing an Ansible playbook, which of the following do you use to refer to a variable named *example*?
- a.** `{{ example }}`
 - b.** `[[example]]`
 - c.** `((example))`
 - d.** `'example'`
- 7.** What is Ansible Vault used for?
- 8.** A playbook is composed of an ordered list of ____.
- a.** tasks
 - b.** plays
 - c.** modules
 - d.** variables
- 9.** True or false: By default, Ansible runs every task on a host in the inventory before moving to the next host.
- a.** True

b. False

10. True or false: Ansible does not support asynchronous tasks.

a. True

b. False

11. A includes a verification task that may not always succeed on the first try. Which Ansible keyword would you use to retry this task several times?

a. *loop*

b. *with_items*

c. *until*

d. *when*

12. When executing a playbook, the default number of forks is five. Which flag can you use to change this value?

a. **-i**

b. **-f**

c. **-u**

d. **-e**

Chapter 5. Using Ansible for Network Automation

This chapter focuses on using Ansible to automate network tasks. So far, all the examples you've seen in this book have had the same focus. This chapter, however, goes into detail about the most common activities network folks carry out.

This chapter shows how to interact with files by reading and writing. It also looks at interactions with several network components—virtual machines, routers, switches, and the cloud—as well as finally APIs.

This chapter finishes with real case studies that show how Ansible has been used to achieve tasks that would otherwise have been very difficult or time-consuming.

You can, of course, use other tools to accomplish the same things you can accomplish with Ansible, but as you will see, using Ansible can give you quick automation wins.

After you read this chapter, you should be able to automate most of your network use cases, with special focus on the ones mentioned in [Chapter 1, “Types of Network Automation.”](#)

Interacting with Files

At one point or another, you must either read from or write to a file. For example, with reporting or monitoring, you gather information and save it for future use. Another example of reading from files is comparing two text configurations when one of them was previously stored in a file. Doing these types of tasks with Ansible is simple, as you will see in the following sections.

Reading

In order to read a file, the first information you need to know is where the file resides. It can be located locally on the control node or remotely in another node. The method you use to read a file depends on the file's location.

To retrieve local files, you can use the *lookup* plug-in, as shown in [Example 5-1](#). This example shows a file (`backup_configs.txt`) from which you can retrieve the contents. From the playbook's execution, you can see the contents of the file displayed.

Example 5-1 Using Ansible to Retrieve a Local File's Contents

```
$ tree
.
├── backup_configs.txt
└── file.yml

$ cat backup_configs.txt
hostname POD4-DC-01
system mtu 9100%

$ cat file.yml
---
- hosts: all

    tasks:
        - debug: msg="the value is {{lookup('file', 'backup_configs.txt')}}"

$ ansible-playbook -i "localhost," file.yml

PLAY [all]
*****
TASK [debug]
*****
ok: [localhost] => {
    "msg": "the value is hostname POD4-DC-01\nsystem mtu 9100"
}
```

```
PLAY RECAP
*****
localhost : ok=1    changed=0    unreachable=0    fa
rescued=0   ignored=0

< >
```

Pretty simple, right? You just have to define the path to the file you want to read; in this case, it is in the same directory, so you simply specify the name. Notice in the output in [Example 5-1](#) that new lines are specified with `\n`.

The *lookup* plug-in does not allow you to retrieve files remotely. You should use *slurp* instead. [Example 5-2](#) shows a playbook for this workflow. There is a file in a remote host (`very_important_file.txt`), and you can retrieve its contents from the local machine by using Ansible's *slurp* module. Note that in this example, the IP address 52.57.45.235 is the remote host, and you would have to use your own remote host and provide its credentials on the inventory file.

Example 5-2 Using Ansible to Retrieve a Remote File's Content

```
admin@52.57.45.235 $ cat very_important_file.txt
this is very important information

$ cat remote_file.yml
---
- hosts: all

  tasks:
    - name: Retrieve remote file
      slurp:
        src: ~/very_important_file.txt
        register: remote_file

    - debug: msg="{{ slurpfile['content'] | b64decode }}" #DECODE

$ ansible-playbook -i inv.yml remote_file.yml
PLAY [all]
```

```
*****
TASK [Retrieve remote file]
*****
ok: [52.57.45.235]

TASK [debug]
*****
ok: [52.57.45.235] => {
    "msg": "this is very important information\n"
}

PLAY RECAP
*****
52.57.45.235 : ok=2     changed=0     unreachable=0
rescued=0    ignored=0
```

As you can see, it is not very complicated to read a remote file. You must specify the remote machine as a host in your hosts file, and you specify the file path on the remote host just as you do for a local lookup. A caveat to keep in mind is that the returned content is in base-64 and must be decoded, as shown in [Example 5-2](#). This is the only possible return value for the *slurp* Ansible module.

Writing

Now that you know how to read from files, you're ready to learn what you do when you want to save information in one.

The easiest way to write to a file is by using the *copy* Ansible module without populating the source variable. You also use this module to copy files. However, you can use the *content* property to write information inside a file. In [Example 5-3](#), the playbook writes all the gathered Ansible facts to file (facts.txt) in the remote host.

Example 5-3 Using Ansible to Write to a Remote File

```

$ cat create_file.yml
---
- hosts: all
  gather_facts: true

  tasks:
    - name: Creating a file with host information
      copy:
        dest: "facts.txt"
        content: |
          {{ ansible_facts }}

```

```

$ ansible-playbook create_file.yml -i "52.57.45.235,"

PLAY [all]
*****

```

```

TASK [Gathering Facts]
*****
ok: [52.57.45.235]

TASK [Creating a file with host information]
*****
changed: [52.57.45.235]

PLAY RECAP
*****
*****
52.57.45.235 : ok=2      changed=1      unreachable=0
skipped=0      rescued=0      ignored=0

```

```

$ cat facts.txt
{"fibre_channel_wwn": [], "module_setup": true, "distribution_version": "Amazon Linux AMI 2018.03.0 HVM", "distribution_file_variety": "Amazon", "env": {"LANG": "C", "TERM": "xterm", "SHELL": "/bin/bash"}, "bash": true, "xdg_runtime_dir": "/run/user/1000", "shlvl": "2", "ssh_tty": true, "python": true}
#####OUTPUT OMITTED#####

```



You can see that you need to specify the destination file path and name—in this example, the default path (which you can omit) and the name facts.txt. Finally, we refer in the *content* field the *ansible_facts* variable. You can see the result of the execution in [Example 5-3](#): Gathered facts are written to the named file in the remote host.

If you want to write to a file on the local machine with information from the remote host, you can register the output in a variable and use the *delegate_to* keyword on the *copy* task. This keyword indicate the host on which to execute the task.

In [Example 5-4](#), Ansible runs *gather_facts* on the remote host and the task *Creating a file with host information* on the localhost.

Example 5-4 Using Ansible to Write to a Local File

```
$ cat create_local_file.yml
---
- hosts: all
  gather_facts: true

  tasks:
    - name: Creating a file with host information
      copy:
        dest: "facts.txt"
        content: |
          {{ ansible_facts }}
      delegate_to: localhost

$ ansible-playbook create_local_file.yml -i "52.57.45.235,"
#####OUTPUT OMITTED####

$ cat facts.txt
{"fibre_channel_wwn": [], "module_setup": true, "distribution_ver
"distribution_file_variety": "Amazon", "env": {"LANG": "C", "TERM"
"SHELL": "/bi
n/bash", "XDG_RUNTIME_DIR": "/run/user/1000", "SHLVL": "2", "SSH_T
```

```
"_": "/usr/bin/python",
#####OUTPUT OMITTED####
```

Although facts.txt has the same content as in [Example 5-3](#), it resides on the local machine (that is, the one where the playbook was run instead of the managed node).

Interacting with Devices

If you are a network engineer, most of your time is probably spent interacting with devices such as routers, switches, firewalls, controllers, servers, and virtual machines. As you have seen in previous chapters, Ansible can connect to most device platforms. Depending on the connection's target, you might need to use a different module. You can connect to these devices to make configuration changes or simply to gather output that may also influence the module used.

If you are not sure what use cases you can automate, refer to [Chapter 1](#), which describes common device interaction use cases.

Networking (Routers, Switches, Firewalls)

As mentioned in [Chapter 4, “Ansible Basics,”](#) networking devices commonly do not support Python. Ansible interacts with devices differently, depending on whether they support Python, but it supports most devices. This section is divided by the method Ansible uses to interact with the devices; SSH, RESTCONF or NETCONF.

Using SSH

As part of your automation workflows, you need data, such as the status of your OSPF neighbors and the RADIUS server connection. Ansible automatically collects some information as part of its initial fact gathering. We can see this by executing a simple playbook against a Cisco Catalyst 9000 switch. You can use the variable *ansible_facts*, as highlighted in

Example 5-5, playbook to print the automatically gathered data.

Example 5-5 Using an Ansible Playbook to Print Host Facts

```
$ cat gather_facts.yaml
---
- name: Ansible Playbook to gather information from IOS Devices
  hosts: ios
  gather_facts: true

  tasks:
    - name: print facts
      debug:
        msg: "{{ ansible_facts }}"
```

Example 5-6 shows an example of output from the playbook's execution. You can see that a lot of information is captured here, although it is not all the information available from the switch. Keep in mind that this example omits some of the output and so does not show all the collected fields.

Example 5-6 Executing an Ansible Playbook to Print Host Facts

```
$ ansible-playbook gather_facts.yaml -i inventory.yml

PLAY [Ansible Playbook to gather information from IOS Devices] ***

TASK [Gathering Facts] ****
ok: [switch_1]

TASK [print facts] ****
ok: [switch_1] => {
  "msg": {
    "discovered_interpreter_python": "/usr/local/bin/python",
    "net_all_ipv4_addresses": [
      "3.6.11.21",
      "192.168.8.54",
      "192.168.8.58",
```

```
"192.168.8.65",
"192.168.8.69",
"192.168.8.73",
"192.168.8.77",
"192.168.0.4"

],
"net_all_ipv6_addresses": [],
"net_api": "cliconf",
"net_filesystems": [
    "flash:"
],
"net_filesystems_info": {
    "flash": {
        "spacefree_kb": 6697044.0,
        "spacetotal_kb": 11087104.0
    }
},
"net_gather_network_resources": [],
"net_gather_subset": [
    "interfaces",
    "default",
    "hardware"
],
"net_hostname": "POD2-GB-01",
"net_image": "flash:packages.conf",
"net_interfaces": {
    "AppGigabitEthernet1/0/1": {
        "bandwidth": 1000000,
        "description": null,
        "duplex": null,
        "ipv4": [],
        "macaddress": "f87b.2053.b2a9",
        "mediatype": "App-hosting port",
        "mtu": 9100,
        "operstatus": "up",
        "type": "App-hosting Gigabit Ethernet"
    ## ## OMITTED OTHER INTERFACES## ##
    }
},
### ## OUTPUT OMITTED## ##
```

```
        }
    }

PLAY RECAP *****
switch_1 : ok=2     changed=0     unreachable=0
rescued=0   ignored=0
```

If you need some specific information that is not captured by *ansible_facts*, you can use the *ios_command* module and specify which command output you want to capture, as shown in [Example 5-7](#). This module is intended for Cisco IOS devices, but there are modules for other vendors and other operating systems. You can also use this approach if you simply need a very small portion of the Ansible gathered facts and you do not want to process all that gathered information. In such a case, you would disable the facts gathering, as also shown in [Example 5-7](#).

Note

Cisco modules are not natively available with Ansible installation. They must be installed manually. One option is for this installation is to use **ansible-galaxy**.

Example 5-7 Using an Ansible Playbook to Capture Output of a *show* Command

```
$ cat ntp.yaml
---
- name: Ansible Playbook to get the NTP status
  gather_facts: no
  hosts: ios

  tasks:
    - name: Get NTP status
      cisco.ios.ios_command:
        commands: show ntp status
```

```

register: ntp_stat

- name: print facts
  debug:
    msg: "{{ ntp_stat }}"

```

[Example 5-8](#) shows an example of the execution of the playbook from [Example 5-7](#). The variable *stdout* stores the contents of the command output.

Example 5-8 Executing the Ansible Playbook to Print NTP Status

```

$ ansible-playbook ntp.yaml -i inventory.yml

PLAY [Ansible Playbook to get the NTP status] ****
TASK [Get NTP status] ****
ok: [switch_1]

TASK [print facts] ****
ok: [switch_1] => {
  "msg": {
    "ansible_facts": {
      "discovered_interpreter_python": "/usr/local/bin/python",
      "changed": false,
      "failed": false,
      "stdout": [
        "Clock is synchronized, stratum 3, reference is 3.6.0.",
        "freq is 250.0000 Hz, actual freq is 249.9977 Hz, precision is 2**1",
        "103916600 (1/100 of seconds), resolution is 4016\nreference time is",
        "(17:23:46.197 UTC Sat Dec 26 2020)\nclock offset is -1.3710 msec",
        "msec\nroot dispersion is 30.90 msec, peer dispersion is 1.09 msec",
        "'CTRL' (Normal Controlled Loop), drift is 0.000009103 s/s\nsystem",
        "last update was 1746 sec ago."
      ],
      "warnings": []
    }
  }
}

```

```
PLAY RECAP ****
switch_1 : ok=2     changed=0     unreachable=0
rescued=0   ignored=0
```

After you get the content in a variable, in this case `stdout`, you could extract any specific needed part by using the parsing techniques described in [Chapter 3, “Using Data from Your Network”](#) (for example, regex).

Tip

When creating regex to capture a specific part of a text string, be mindful of trying it on all possibilities that the original source might output. Regex are sensitive, and failing to account for all possibilities happens often, especially for corner cases.

Now you know how to gather information from a device by using Ansible, but how do you configure one? You use the configuration module. To see how this is done, let’s look at the configuration of an interface in a Cisco Catalyst 9000. [Example 5-9](#) uses the `ios_config` module to send two CLI commands (**description** and **speed**). Furthermore, it uses the *parents* keyword to instruct the module to insert these commands under the interface TenGigabitEthernet1/1/3.

Example 5-9 Ansible Playbook to Configure an IOS Interface by Using `ios_config`

```
---
- name: Ansible Playbook to configure an interface
  hosts: ios

  tasks:
    - name: configure interface
      cisco.ios.ios_config:
        lines:
```

```
- description Ansible interface
- speed 1000
parents: interface TenGigabitEthernet1/1/3
```

Note

The *ios_config* module is intended for Cisco IOS devices, just as *ios_command* is. There are equivalent modules for other vendors and software. Ansible has modules for many networking vendors, including Cisco, Juniper, Arista, and F5.

Using the *ios_config* module is a generic way of achieving a configuration, and you must provide the CLI commands you want to execute. There are specific modules for some configurations, and you can configure specific features without knowing the CLI syntax required. The playbook in [Example 5-10](#) achieves the same goal as the one in [Example 5-9](#), configuring a description and a specific speed for the interface TenGigabitEthernet1/1/3; however, it uses a different module, the *ios_interface* module. With this module, you do not pass the CLI commands but simply pass variables to the Ansible module.

Example 5-10 Ansible Playbook to Configure an IOS Interface Using *ios_interface*

```
---
- name: Ansible Playbook to configure an interface
hosts: ios

tasks:
- name: Replaces interface configuration
  cisco.ios.ios_interfaces:
    config:
      - name: TenGigabitEthernet1/1/3
        description: Ansible interface
        speed: 1000
    state: replaced
```

Note

If a module is available for your use case, it is typically a good idea to use it due to the abstraction it provides from the CLI syntax, which can be different for different software versions.

Using NETCONF

So far, all the Ansible modules covered in this chapter for configuring and retrieving data from network devices have used SSH. If you have already adopted or are in the process of adopting newer management protocols, such as NETCONF (refer to [Chapter 2, “Data for Network Automation”](#)), you can still use Ansible as your automation tool.

When you use NETCONF, there are the three modules to consider:

- *netconf_get*
- *netconf_config*
- *netconf_rpc*

The *netconf_get* Ansible module allows you to retrieve data from NETCONF-enabled devices. It is worth noting that this module can display the results in multiple data formats (for example, JSON, XML). The playbook shown in [Example 5-11](#) attempts to retrieve the running configuration of the interface GigabitEthernet1/0/13 and display it in JSON format. This example uses the same Cisco Catalyst 9000 device accessed with SSH earlier in this chapter.

Example 5-11 Ansible Playbook to Retrieve the Running Configuration Using NETCONF

```
$ cat nc_get_running.yaml
---
- hosts: all
  connection: netconf
  gather_facts: no
```

```

tasks:
  - name: Get configuration and state data from the running data
    ansible.netcommon.netconf_get:
      source: running
      filter: <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf
interfaces"><interface><name>GigabitEthernet1/0/13</name></interfa
      display: json
      register: result

  - name: Print the running configuration
    ansible.builtin.debug:
      msg: "{{ result }}"

```

< **>**

[Example 5-12](#) shows an example execution of the playbook from [Example 5-11](#). It is worth noting that the playbook is completely agnostic of the network device you are accessing, as long as it has a running datastore and the corresponding YANG module used in the filter.

Example 5-12 Executing the Ansible Playbook to Retrieve the Running Configuration Using NETCONF

```

$ ansible-playbook nc_get_running.yaml -i inventory.yml
PLAY [all]
*****
TASK [Get configuration and state data from the running datastore]
*****
ok: [10.78.54.124]

TASK [Print the running configuration]
*****
ok: [10.78.54.124] => {
  "msg": {
    "ansible_facts": {
      "discovered_interpreter_python": "/usr/bin/python"
    },

```

```

"changed": false,
"failed": false,
"output": {
    "data": {
        "interfaces": {
            "interface": {
                "enabled": "true",
                "ipv4": "",
                "ipv6": "",
                "name": "GigabitEthernet1/0/13",
                "type": "ianaift:ethernetCsmacd"
            }
        }
    }
},
"stdout": "",
"stdout_lines": [],
"warnings": []
}
}

PLAY RECAP
*****
10.78.54.124 : ok=2      changed=0      unreachable=0
rescued=0      ignored=0
< >

```

The *netconf_config* Ansible module, as its name indicates, is used to make configuration changes. All you must do is provide the configuration in the *content* property, as shown in [Example 5-13](#). This example sets the new IP address 10.3.0.1/24 and adds a new description for the interface GigabitEthernet1/0/13.

Example 5-13 Ansible Playbook to Configure an Interface Using NETCONF

```

---
- hosts: all
  connection: netconf

```

```

gather_facts: no

tasks:
  - name: configure interface IP and description
    ansible.netcommon.netconf_config:
      content: |
        <config>
          <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
            <interface>
              <name>GigabitEthernet1/0/13</name>
              <description>new description</description>
              <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
                <address>
                  <ip>10.3.0.1</ip>
                  <netmask>255.255.255.0</netmask>
                </address>
              </ipv4>
            </interface>
          </interfaces>
        </config>

```

Note

Keeping your YANG configurations outside your playbooks in separate files is recommended as it improves a playbook's readability.

You may need to perform actions related to NETCONF that are neither configuration nor data retrieval actions. This is where the *netconf_rpc* Ansible module comes in. It is a generic module that is capable of calling different types of remote procedure calls (RPCs). The following are some examples of possible actions:

- Copying the running configuration to the startup configuration
- Retrieving schemas

- Locking/unlocking datastores

In the networking world, one the most common actions is saving a configuration in NVRAM. You can achieve this with the playbook shown in [Example 5-14](#). This playbook is built for IOS XE, and a different device type might require a different *xmlns* value, which is the *XML namespace* (refer to [Chapter 2](#)).

Example 5-14 Ansible Playbook to Save the Running Configuration Using NETCONF

```
---
- hosts: all
  connection: netconf
  gather_facts: no

  tasks:
    - name: copy running to startup
      ansible.netcommon.netconf_rpc:
        xmlns: "http://cisco.com/yang/cisco-ia"
        rpc: save-config
```

By using this module, you can specify any type of RPC method, including *get*. However, you should use it only when your action is not defined in a more specific module, such as the previously mentioned *netconf_config* and *netconf_get* modules.

Using RESTCONF

Another next-generation management protocol introduced in [Chapter 2](#) is RESTCONF. If this is the protocol you are using to manage your devices, you are in luck because it is also fully supported by Ansible modules.

There are two important RESTCONF modules to keep in mind, and their names are very similar to the names of the NETCONF modules:

- *restconf_get*
- *restconf_config*

You use `restconf_get` to obtain information from RESTCONF-enabled devices. You cannot specify any HTTP verb with this module; the only one supported is GET.

[Example 5-15](#) shows a playbook that retrieves the configuration of the interface GigabitEthernet1/0/13. It does so by specifying the URI in the `path` variable, the output format in the `output` variable, and the type of data to retrieve in the `content` variable (which can take the value `config`, `nonconfig`, or `all`).

Note

For a refresher on URI encoding rules for RESTCONF, refer to [Chapter 2](#).

Example 5-15 Ansible Playbook to Retrieve a Configuration Using RESTCONF

```
$ cat rc_get.yaml
---
- hosts: all
  connection: httpapi
  gather_facts: no

  vars:
    ansible_httpapi_port: 443
    ansible_httpapi_use_ssl: yes
    ansible_network_os: restconf
    ansible_httpapi_validate_certs: false

  tasks:
    - name: Get Interface Ge1/0/13 Configuration
      restconf_get:
        content: config
        output: json
        path: /data/ietf-interfaces:interfaces/interface=GigabitEt
      register: cat9k_rest_config
```

```
- debug: var=cat9k_rest_config
```

There are plenty of interesting things to notice in the playbook in [Example 5-15](#). One of them is that the connection specified is not RESTCONF but *httpapi*, which tells Ansible to use HTTP. The combination of this connection type with the variable *ansible_network_os* being assigned the value *restconf* is what defines a RESTCONF connection for a host.

You have probably noticed three other variables related to the *httpapi* connection type. You can specify them according to your needs in your inventory file for each host. This is done directly in the playbook in [Example 5-15](#) for ease of visualization only. These variables mandate the use of HTTPS using port 443 but do not validate the certificates in use as this example uses self-signed certificates. [Example 5-16](#) shows the execution of this playbook on a single device, a Catalyst 9000. You can see that, by default, the module *restconf_get* does not return anything (note the task *Get Interface Ge1/0/13 Configuration*). If you want to see output, you must register the result to a variable and print it; as you have seen in previous examples, you can use the *register* and *debug* keywords for this.

Example 5-16 Executing the Ansible Playbook to Retrieve a Configuration Using RESTCONF

```
$ ansible-playbook rc_get.yml -i inventory.yml

PLAY [all] ****
TASK [Get Interface Ge1/0/13 Configuration]
*****
ok: [10.78.54.124]

TASK [debug] ****
ok: [10.78.54.124] => {
    "cat9k_rest_config": {
        "ansible_facts": {
            "discovered_interpreter_python": "/usr/bin/python"
    }
}
```

```

        },
        "changed": false,
        "failed": false,
        "response": {
            "ietf-interfaces:interface": {
                "enabled": true,
                "ietf-ip:ipv4": {},
                "ietf-ip:ipv6": {},
                "name": "GigabitEthernet1/0/13",
                "type": "iana-if-type:ethernetCsmacd"
            }
        },
        "warnings": []
    }
}

PLAY RECAP ****
10.78.54.124 : ok=2      changed=0      unreachable=0
rescued=0      ignored=0
< >

```

Building on the previous examples, you can use the *restconf_config* module to configure, edit, or delete values on a RESTCONF-enabled device. This module is more versatile than the *restconf_get* module, as you can specify HTTP verbs such as the following:

- POST
- PUT
- PATCH
- DELETE

Note

[Chapter 2](#) describes use cases for the HTTP verbs.

[Example 5-17](#) shows how to configure the interface GigabitEthernet1/0/13

with the IP address 10.0.0.1/24 and a description. From the playbook's execution, you can see the difference from [Example 5-16](#), where the interface was empty. Because this example uses the HTTP verb PATCH, every nonspecified previously configured attribute remains unchanged.

Example 5-17 Ansible Playbook to Configure a Device Using RESTCONF

```
$ cat rc_configure.yaml
---
- hosts: all
  connection: httpapi
  gather_facts: no

vars:
  ansible_httpapi_port: 443
  ansible_httpapi_use_ssl: yes
  ansible_network_os: restconf
  ansible_httpapi_validate_certs: false

tasks:
  - name: Configure IP address
    ansible.netcommon.restconf_config:
      path: /data/ietf-interfaces:interfaces/interface=GigabitEthernet1
      method: patch
      content: |
        {
          "ietf-interfaces:interface": {
            "name": "GigabitEthernet1/0/13",
            "ietf-ip:ipv4": {
              "address": [
                {
                  "ip": "10.0.0.1",
                  "netmask": "255.255.255.0"
                }
              ]
            }
          "description": "New description"
        }
```

```

        }

- name: Get Interface Ge1/0/13 Configuration
  restconf_get:
    content: config
    output: json
    path: /data/ietf-interfaces:interfaces/interface=GigabitEthernet1
  register: cat9k_rest_config

- debug: var=cat9k_rest_config

$ ansible-playbook rc_configure.yaml -i inventory.yml

PLAY [all] ****
TASK [Configure IP address]
*****
changed: [10.78.54.124]

TASK [Get Interface Ge1/0/13 Configuration]
*****
ok: [10.78.54.124]

TASK [debug] ****
ok: [10.78.54.124] => {
  "cat9k_rest_config": {
    "changed": false,
    "failed": false,
    "response": {
      "ietf-interfaces:interface": [
        {
          "description": "New description",
          "enabled": true,
          "ietf-ip:ipv4": [
            {
              "address": [
                {
                  "ip": "10.0.0.1",
                  "netmask": "255.255.255.0"
                }
              ]
            }
          ]
        }
      ]
    }
  }
}

```

```
        },
        "ietf-ip:ipv6": {},
        "name": "GigabitEthernet1/0/13",
        "type": "iana-if-type:ethernetCsmacd"
    }
}
}

PLAY RECAP ****
10.78.54.124 : ok=3      changed=1      unreachable=0
rescued=0      ignored=0
```

Tip

If you have doubts or difficulties creating content payloads, keep in mind that an easy way is to configure what you need manually on a device and execute a GET operation. You can use the result to configure any other device.

Computing (Servers, Virtual Machines, and Containers)

Computing is a broad domain encompassing a big range of platforms, in this section we divide in two major sections: Servers and Virtual machines, and Containers.

For Servers and Virtual machines, we will cover actions executed directly on them (such as installing software), but also interactions with the platforms that support the virtualization, for example VMware vCenter. The same happens for containers, where we cover individual container technologies like Docker and container orchestration platforms like Kubernetes.

For each section we show you example playbooks to interact with these components and sample executions of those playbooks.

Servers and Virtual machines

Servers are part of networks, and although they are not always managed by the network team, there is always a server to connect to. Servers come in different flavors, and they use different syntax, which can be a management burden for someone who is not a seasoned systems administrator. Common server-related tasks include installing software, collecting outputs, and expanding storage space. Ansible can help you address servers in a scalable manner.

When working with servers, you can use modules for the package manager of your system (for example, **yum**, **apt-get**). [Example 5-18](#) shows a playbook that installs the latest version of httpd by using the **yum** Ansible module.

Example 5-18 Ansible Playbook to Install httpd Using the **yum** Module

```
$ cat httpd.yml
---
- name: Install the latest version of Apache httpd
  yum:
    name: httpd
    state: latest
```

In the event that you need to do an operation that is not available in a module, you can use the *command* module and specify the command you need in its raw form. This is similar to the generic *ios_config* module you saw earlier in this chapter, but it is for servers instead of network devices.

The playbook in [Example 5-19](#) does the same thing as the one in [Example 5-18](#): It installs httpd. However, it uses the *command* module instead of the *yum* module. In a typical scenario, you would not use the *command* module because *yum* exists as an Ansible module, and it is the preferred option.

Example 5-19 Ansible Playbook to Install httpd Using the *command* Module

```
$ cat httpd.yml
---
- hosts: all
```

```
tasks:  
  - name: Install httpd using yum  
    command: yum install httpd  
    register: result  
  
  - debug:  
    msg: "{{ result }}"
```

[Example 5-20](#) shows an example of the execution of the playbook in [Example 5-19](#). Notice that the return message indicates that httpd is already installed. This is because the playbooks in [Example 5-18](#) and [Example 5-19](#) have already been run on the same host. You can see from this example the intelligence that is embedded in playbooks by default. If you ran the playbook in [Example 5-19](#) and http had not been installed before, *stdout* would show the installation log.

Note

To execute the examples in this section, you need a Linux host defined in your inventory file.

Example 5-20 Executing the Ansible Playbook to Install httpd Using the command Module

```
$ ansible-playbook -i inv.yml httpd.yml  
  
PLAY [all]  
*****  
  
TASK [Gathering Facts]  
*****  
ok: [10.10.10.1]  
  
TASK [Install httpd using yum]  
*****
```

```

changed: [10.10.10.1]

TASK [debug]
*****
ok: [10.10.10.1] => {
    "msg": {
        "changed": true,
        "cmd": [
            "yum",
            "install",
            "httpd"
        ],
        "delta": "0:00:00.720593",
        "end": "2020-12-27 09:33:49.812601",
        "failed": false,
        "rc": 0,
        "start": "2020-12-27 09:33:49.092008",
        "stderr": "",
        "stderr_lines": [],
        "stdout": "Loaded plugins: extras_suggestions, langpacks,
motd\nPackage httpd-2.4.46-1.amzn2.x86_64 already installed and la
to do",
        "stdout_lines": [
            "Loaded plugins: extras_suggestions, langpacks, priori
            "Package httpd-2.4.46-1.amzn2.x86_64 already installed
            "Nothing to do"
        ],
        "warnings": []
    }
}

PLAY RECAP
*****
10.10.10.1 : ok=3      changed=1      unreachable=0      f
rescued=0    ignored=0

```

So far, the playbooks in Examples 5-18 and 5-19 have shown you how to

automate the installation of *httpd*, but you still need to know which type of operating system you are connecting to in order to issue the correct command for the corresponding package manager. This can be a management burden.

The playbook in [Example 5-21](#) addresses this potential burden by collecting information about the operating system family (using *ansible_facts*) and uses this data to decide which package manager to use.

Note

[Example 5-21](#) shows a useful application of the *when* statement, which is covered in the [Chapter 4](#).

Example 5-21 Ansible Playbook to Install httpd Independently of the Operating System

```
---
- hosts: all
  gather_facts: true

  tasks:
    - name: Install the latest version of Apache on RedHat
      yum:
        name: httpd
        state: latest
      when: ansible_os_family == "RedHat"

    - name: Install the latest version of Apache on Debian
      apt:
        name: httpd
      when: ansible_os_family == "Debian"
```

A playbook like the one in [Example 5-21](#) would be created once in a collaboration between you and the computing team. In the future, you would not have to worry again about different operating system syntaxes when executing changes.

Many of the facts that Ansible collects from computing systems are useful in common system administrator tasks, including the following examples:

- Mounted disk utilization
- Virtual memory available
- Network interface properties

You can use these metrics to derive actions, such as mounting another virtual disk or deleting files when you are running out of space. [Example 5-22](#) shows an example of a playbook that achieves this.

Example 5-22 Using Ansible to Verify and Possibly Free Disk Space

```
---
- hosts: all
  gather_facts: true

  tasks:
    - name: Verify available disk space
      assert:
        that:
          - item[0].mount != '/' or {{ item[0].mount == '/' and
> (item[0].size_total|float * 0.5) }}
      loop:
        - "{{ ansible_mounts }}"
      ignore_errors: yes
      register: disk_free

    - name: free disk space
      command: "rm -rf SOMEFILE"
      when: disk_free is failed
```

Let us go over what the playbook in [Example 5-22](#) does. The first task iterates over the mounted volumes (gathered using Ansible facts) and asserts whether the size available in the root partition is over 50%. You must ignore errors on this task because you do not want the playbook to stop if the assertion is false. You want to register the result to a variable and free disk

space on the second task if more than 50% is occupied.

So far, all the examples in this section, have involved connecting to operating systems directly. However, plenty of computing tasks revolve around infrastructure virtualization. Ansible is a great tool for automating these tasks.

Common tasks in VMware environments include the following:

- Creating/modifying virtual machines
- Creating/modifying port groups
- Configuring virtual/distributed switches
- Performing VMotion across hosts
- Creating templates and/or snapshots

Example 5-23 shows an Ansible playbook that creates a new virtual machine (VM) from a template. This template, *template-ubuntu*, would need to have been created previously. A VM template is a copy of a virtual machine that includes VM disks, virtual devices, and settings. Using one is a great way to replicate virtual machines; however, how to use VM templates is beyond the scope of this book.

Example 5-23 Ansible Playbook to Deploy a VM from a VM Template

```
---
- hosts: localhost
  gather_facts: no
  vars:
    vcenter_server: "10.1.1.1"
    vcenter_user: "administrator@vsphere.local"
    vcenter_pass: "password"
    datacenter_name: "Datacenter"
    cluster_name: "Cluster01"

  tasks:
    - name: Clone the template
      vmware_guest:
        hostname: "{{ vcenter_server }}"
        username: "{{ vcenter_user }}"
```

```

password: "{{ vcenter_pass }}"
validate_certs: False
name: web
template: template-ubuntu
datacenter: "{{ datacenter_name }}"
folder: /{{ datacenter_name }}/vm
cluster: "{{ cluster_name }}"
datastore: "iscsi-datastore01"
networks:
- name: VM Network
  ip: 10.0.0.5
  netmask: 255.255.255.0
  gateway: 10.0.0.254
  type: static
  dns_servers: 10.0.0.1
state: poweredon
wait_for_ip_address: yes
delegate_to: localhost

```

[Example 5-23](#) uses the module `vmware_guest` to create the VM. Like the Cisco Ansible module, it must be installed manually. This module is paramount for the VMware environment because it can also reconfigure VMs, delete VMs, or trigger actions (for example, reboots, restarts, suspends). The parameter that defines what is to be executed is `state`. There are many possible states; [Table 5-1](#) lists and describes them.

Table 5-1 `vmware_guest State Choices`

State	Description
<i>absent</i>	If the virtual machine exists, the virtual machine and its components are removed.
<i>poweredon</i>	If the virtual machine exists but is in any state other than powered on, the virtual machine changes state to powered on. If it does not exist, it is created in this state.
<i>poweredoff</i>	If the virtual machine exists but is in any state other than powered off, the virtual machine changes state to powered off. If it does not exist, it is created in this state.
<i>present</i>	If the virtual machine exists, it verifies that the virtual machine configurations conform with the given parameters. If it does not exist, it is created.
<i>rebootguest</i>	If the virtual machine exists, it is rebooted.
<i>restarted</i>	If the virtual machine exists, it is restarted.
<i>suspended</i>	If the virtual machine exists, it is suspended.
<i>shutdownguest</i>	If the virtual machine exists, it is shut down.

This example defines the authentication credentials on the playbook itself for ease of understanding. However, in a production scenario, you would separate the credentials from the playbook (using something like Ansible Vault, which is covered in [Chapter 4](#)). The playbook in [Example 5-23](#) specifies the virtual machine name (*web*), along with the data center (*Datacenter*), the folder within the data center, the cluster (*Cluster01*), the datastore (*iscsi-datastore01*), and some guest network information. Notice that this module runs locally on the control node.

After the VM is created, you can continue automating by connecting to the VM using the defined IP address and provisioning it using another task in the same playbook. Or you can trigger a snapshot, as shown in [Example 5-24](#).

Example 5-24 Ansible Playbook to Trigger a VM Snapshot and Perform VMotion on It

```
---
- hosts: localhost
  gather_facts: no
```

```

vars:
  vcenter_server: "10.1.1.1"
  vcenter_user: "administrator@vsphere.local"
  vcenter_pass: "password"
  datacenter_name: "Datacenter"
  cluster_name: "Cluster01"

tasks:
- name: Create VM snapshot
  vmware_guest_snapshot:
    hostname: "{{ vcenter_server }}"
    username: "{{ vcenter_user }}"
    password: "{{ vcenter_pass }}"
    datacenter: "{{ datacenter_name }}"
    folder: /{{ datacenter_name }}/vm
    name: web
    state: present
    snapshot_name: snap1
    description: snapshot_after_creation
  delegate_to: localhost

- name: Perform vMotion of virtual machine
  vmware_vmotion:
    hostname: '{{ vcenter_server }}'
    username: '{{ vcenter_user }}'
    password: '{{ vcenter_pass }}'
    vm_name: web
    destination_host: ESXI01
  delegate_to: localhost

```

Although [Example 5-24](#) is a playbook on its own, you can see from the variables in use that it targets the VM created in [Example 5-23](#). [Example 5-24](#) creates a VM snapshot of the web VM named snap1 and then performs VMotion to a different host (ESXI01).

Note

Although the focus here is VMware, there are plenty of Ansible

modules for other virtualization platforms, including Red Hat Virtualization (RHV), XenServer, OpenStack, and Vagrant.

Containers

Another virtualization technique involves the use of containers. Containerization is one of the latest trends in the industry.

We do not go into detail about what containers are or why would you use them. We focus here on how to use virtualization with them. If you are using Docker containers, you can manage them with Ansible. Ansible is not the most appropriate tool for this, as there are better container orchestration solutions, but it can be useful when you are developing end-to-end playbooks.

[Example 5-25](#) shows how to create an Nginx container named `web_server` by using the `docker_container` Ansible module.

Example 5-25 Ansible Playbook to Create an Nginx Container

```
---
- hosts: all

  tasks:
    - name: Run a nginx container
      docker_container:
        name: web_server
        image: nginx
```

You can do other Docker-related tasks such as building containers and pushing them to a repository, as shown in [Example 5-26](#). This example shows how to retrieve a Dockerfile from the `./nginx` path, build the container, and push it to the repository `repo/nginx` with the tag `v1`.

Example 5-26 Ansible Playbook to Build an Nginx Container

```
---
```

```

- hosts: all

tasks:
  - name: Build an image and push
    docker_image:
      build:
        path: ./nginx
        name: repo/nginx
        tag: v1
      push: yes
      source: build

```

If you are using containers but are not using Docker directly and instead are using an orchestrator such as Kubernetes, you need different Ansible modules. [Example 5-27](#) shows a playbook that uses the `k8s` module to create a namespace, if it is not already present, and deploys two Nginx containers in that newly created namespace in a Kubernetes cluster.

Example 5-27 Ansible Playbook to Create an Application in a Specific Namespace

```

---
- hosts: all

tasks:
  - name: create namespace
    k8s:
      name: my-namespace
      api_version: v1
      kind: Namespace
      state: present
  - name: deploy a web server
    k8s:
      api_version: v1
      namespace: my-namespace
      definition:
        kind: Deployment
      metadata:

```

```
labels:  
    app: nginx  
    name: nginx  
spec:  
    replicas: 2  
    selector:  
        matchLabels:  
            app: nginx  
    template:  
        metadata:  
            labels:  
                app: nginx  
        spec:  
            containers:  
                - name: my-webserver  
                  image: repo/nginx
```

Note

If you are not familiar with the Kubernetes naming convention, [Example 5-27](#) might look intimidating. From an Ansible perspective, it is just another module where you must populate the needed variables.

Cloud (AWS, GCP)

As mentioned in [Chapter 1](#), the cloud is often an extension of networks today. For some enterprises, the public cloud is their whole network, and they have decommissioned their on-premises equipment. Netflix is an example of a customer that solely uses the public cloud.

Ansible can also help you manage the cloud, although Terraform would be a better fit due to its focus on provisioning. These tools come in handy especially when you have several cloud providers and want to abstract the syntax from the operator, just as we did for operating systems earlier in this chapter.

This section focuses on interacting with cloud components. If your goal is interacting with components in the public cloud—for example, a virtual machine or a file—you can use the same syntax and playbooks shown for those component types in previous sections of this chapter.

Each public cloud provider has a different virtual machine service. For example, if you want to create a VM in Amazon Web Services (AWS), the service is called EC2, in Google Cloud Platform (GCP), the service is called Compute Engine, and in Microsoft Azure, it is called Virtual Machines.

[Example 5-28](#) shows how to create a virtual machine in AWS by using the EC2 service. This example only allows the user to choose the subnet where the VM should be created, along with the AMI (Amazon machine image), which is the operating system. In this example, the AMI *ami-03c3a7e4263fd998c* represents Amazon Linux 2 AMI (64-bit x86).

You can modify this example and parametrize the inputs you want the operator to be able to change. The subnet, AMI, and key must already exist in your cloud environment in order to be referenced by the playbook.

Note

Having playbooks with static values helps achieve standardization and compliance (such as when all your infrastructure must have some type of characteristic—for example, the same access key). Standardization and compliance are very important for maintaining a manageable cloud environment.

Example 5-28 Ansible Playbook to Create an EC2 Virtual Machine

```
---
- hosts: all
  vars:
    subnet: "subnet-72598a0e"
    ami: "ami-03c3a7e4263fd998c"

  tasks:
    - amazon.aws.ec2:
```

```
key_name: key
instance_type: t2.micro
image: "{{ ami }}"
wait: yes
vpc_subnet_id: "{{ subnet }}"
assign_public_ip: yes
```

Note

This type of playbook runs locally from your control node.

Ansible has modules available for all major cloud providers. [Example 5-29](#) shows a playbook that creates a 50 GB storage disk for GCP. As in the EC2 case in [Example 5-28](#), all referenced resources in [Example 5-29](#) (*source_image*, *zone*, *project*, *serviceaccount*) must already exist, or the playbook's execution will fail.

Example 5-29 Ansible Playbook to Create a GCP Disk

```
---
- hosts: all
  vars:
    gcp_cred_file: "/tmp/test-project.json"

  tasks:
    - name: create a disk
      gcp_compute_disk:
        name: disk-instance01
        size_gb: 50
        source_image: projects/ubuntu-os-cloud/global/images/fam
        zone: "us-central1-a"
        project: "633245944514"
        auth_kind: "serviceaccount"
        service_account_file: "{{ gcp_cred_file }}"
        state: present
      register: disk
```

You can fully provision a cloud environment by chaining multiple tasks in an Ansible playbook or executing several playbooks in a row. [Example 5-30](#) expands on [Example 5-28](#) by creating an SSH key and firewall rules before creating the virtual machine on AWS. Furthermore, it saves the newly created key (`key-private.pem`) locally so you can access the virtual machine later.

Example 5-30 Ansible Playbook to Provision Resources on AWS

```
---
- hosts: all
  vars:
    subnet: "subnet-72598a0e"
    ami: "ami-03c3a7e4263fd998c"

  tasks:
    - name: Create an EC2 key
      ec2_key:
        name: "ec2-key"
      register: ec2_key

    - name: Save private key
      copy: content="{{ ec2_key.key.private_key }}" dest=".//key-pr
      when: ec2_key.changed

    - name: Create security group
      ec2_group:
        name: "ec2_security_group"
        description: "ec2 security group"
        rules:
          - proto: tcp  # ssh
            from_port: 22
            to_port: 22
            cidr_ip: 0.0.0.0/0
          - proto: tcp  # http
            from_port: 80
            to_port: 80
            cidr_ip: 0.0.0.0/0
```

```
- proto: tcp  # https
  from_port: 443
  to_port: 443
  cidr_ip: 0.0.0.0/0
rules_egress:
- proto: all
  cidr_ip: 0.0.0.0/0
register: firewall

- amazon.aws.ec2:
  key_name: ec2-key
  instance_type: t2.micro
  image: "{{ ami }}"
  wait: yes
  vpc_subnet_id: "{{ subnet }}"
  assign_public_ip: yes
  group_id: "{{ firewall.group_id }}"
```

When you and your team use playbooks, you no longer need to access the clouds' graphical interfaces, and you reduce human error and increase speed and agility. Automation is a cloud best practice that is recommended by all cloud vendors.

Note

Keep in mind that Terraform is also a great tool for cloud resources provisioning.

Interacting with APIs

Chapter 2 covers APIs and their value when it comes to gathering data. Ansible can interact with APIs to enable you to automate actions based on collected data.

Some modules automatically use APIs as the connection mechanism (for

example, the Cisco *FTD* module). However, if you need to connect to custom APIs, you can use the *uri* module. It is a built-in module in Ansible that facilitates interaction with HTTP/S web services.

[Example 5-31](#) shows an API that requires you to acquire a token with your user credentials and use that token in your future requests. It uses the DNA Center API in a sandbox environment.

Example 5-31 Ansible Playbook to Get All Network Devices from DNAC

```
---
- hosts: all

  tasks:
    - name: Get login Token
      uri:
        url: "https://sandboxdnac.cisco.com/api/system/v1/auth/token"
        method: POST
        user: "devnetuser"
        password: "Cisco123!"
        force_basic_auth: yes
      register: token_response

    - name: Get network devices
      uri:
        url: "https://sandboxdnac.cisco.com/api/v1/network-device"
        method: GET
        headers:
          x-auth-token: "{{ token_response.json.Token }}"
      register: devices_response

    - name: print response
      debug:
        msg: "{{ devices_response }}"
```

Note

[Chapter 2](#) shows this same API call with **curl**.

The first task in [Example 5-31 \(Get login Token\)](#) issues a POST request with the user credentials. The second task (*Get network devices*) uses the returned token in the headers to issue a GET request for configured devices on the system.

Try out this example yourself by saving the playbook in [Example 5-31](#) as API.yml and executing the following command (with SSH enabled on your localhost):

```
$ ansible-playbook -i "localhost," API.yml
```

You can also use this same module, *uri*, to interact with websites and crawl their content, even though websites are not APIs. The playbook in [Example 5-32](#) is similar to the one in [Example 5-31](#), but instead of using an API URL, it provides the URL of Cisco's website. You can execute this example by using the same syntax as previously mentioned. In the content variable named *result*, you can see an HTML text representation document of the page.

Example 5-32 Ansible Playbook to Crawl Website Content

```
---
- hosts: all
  gather_facts: no

  tasks:
    - name: Crawl Cisco.com website
      uri:
        url: https://www.cisco.com
        return_content: yes
      register: result

    - name: Print content
      debug:
        msg: "{{ result }}"
```

The preferred way to crawl web pages is actually by using the *lookup* function, as shown in [Example 5-33](#), instead of the *uri* module.

Example 5-33 Ansible Playbook to Crawl Website Content Using *lookup*

```
---
- hosts: all
  gather_facts: no

  tasks:
    - name: Crawl Cisco.com website
      debug: msg="{{ lookup('url', 'https://www.cisco.com') }}"
```

Note

You run API playbooks run from your control node.

Although you have seen ways of building playbooks to connect to custom API endpoints, the preferred way is to use modules, as shown throughout this chapter. Use the *uri* module only when strictly necessary.

To spark your use case imagination, the following are some of the common components in the networking world that have modules available:

- GitHub
- ServiceNow
- Jenkins
- Splunk
- Slack

Case Studies

This section presents three case studies based on real customer requests. They show how Ansible can be used to make configuration changes and enable

global-scale automation.

Some of these use cases have code examples, but note that these use cases are simply combinations of the techniques you have learned in this chapter and in [Chapter 4](#). This section shows how you can solve global-scale networking challenges by intelligently combining Ansible modules into playbooks.

Configuration Changes Across 100,000 Devices

Customer XYZ has an installed base throughout Europe that entails nearly 100,000 devices. This network has been running for well over 10 years, and it includes a multitude of different vendors and platforms. It also includes the following components:

- Firewalls
- Wireless LAN controllers
- Switches and routers
- Servers

All these devices are accessed using TACACS authentication.

XYZ's security team identified a necessity: It needed to modify the TACACS key, as it was the same for every device and had never changed since the network was initially created. Also, the team thought it would be good to be able to rotate this key every 6 months. They were facing European compliance issues, so these necessities had to be addressed.

We were involved in automating the process, as manually changing so many thousands of devices did not seem feasible. Ansible was chosen as the automation tool.

The workflow we developed had many stages, including a number of verifications. Losing access to one of these devices involved a repair cost because some of them were in remote locations and not easily accessible. The workflow steps were as follows:

Step 1. Collect the device configuration.

Step 2. Validate the TACACS authentication.

Step 3. Parse the source IP address for the TACACS server.

Step 4. Modify the TACACS key on the device.

Step 5. Modify the device entry on the TACACS server, using the source IP address.

Step 6. Validate the TACACS authentication.

We used a different connection mechanism for each component: We had Ansible modify the TACACS server by using an API, and we used SSH for the devices.

The biggest difficulties we faced were related to the sparseness of the installed base. The playbook had to identify what device type it was connecting to in order to be able to correctly make the change. For this, we used a dynamic inventory script that parsed a third-party device inventory database containing information regarding what operating system was running on each device.

We successfully changed all devices in record time: one month. By using the same playbooks, XYZ is now able to rotate its keys whenever necessary.

The playbook in [Example 5-34](#) is a scaled-down version of the playbooks we used. It is specifically for NX-OS. This example can be seen as an end-to-end example that touches several components.

We highlight the main tasks in [Example 5-34](#): changing the TACACS configuration on the device and on the TACACS server. However, you can see that most of the beginning of the playbook uses regex along with gathered facts to identify relevant information (for example, the configured source IP address) that is required for these key tasks that occur later in the playbook.

Example 5-34 Ansible Playbook to Update TACACS Key for NX-OS

```
- name: Ansible Playbook to change TACACS keys for NXOS Devices
  hosts: nxos
  connection: local
  gather_facts: yes
  vars:
    tacacs_key: 'VerySecretKey1'
```

```

    tacacs_server_ip1: "10.0.0.1"
    tacacs_source_regex: "{{ 'ip tacacs source-interface ([aA-zZ]+\\-?[aA-zZ]+\\d\\/\\/?\\d?\\/\\/?\\d?\\/?:?:?\\d?\\.?.\\d?\\d?\\d?\\d?)'

roles:
  - TACACSrole

tasks:
  - name: Scan running-config for TACACS information
    set_fact:
      tacacs_source_interface: "{{ ansible_facts.net_config | regex.findall(tacacs_source_regex, multiline=True) }}"

  - name: Retrieve TACACS VRF section
    cisco.nxos.nxos_command:
      commands: show running-config | section ^aaa
      register: conf_aaa

  - name: Retrieve TACACS VRF line
    set_fact:
      conf_aaa_vrf: "{{ conf_aaa.stdout[0] | regex.findall('use-v multiline=True) }}"

  - name: Register TACACS VRF
    set_fact:
      conf_aaa_vrf_name: "{{ (conf_aaa_vrf | first).split()[1] }}"
      when: conf_aaa_vrf | length > 0

  - name: Assert that 'ip tacacs source-interface is configured'
    assert:
      that:
        - "{{ tacacs_source_interface | length }} == 1"

  - name: Scan running-config for tacacs source-interface IP add
    set_fact:
      tacacs_source_ip: "{{ ansible_facts.net_interfaces[tacacs_source_interface[0]].ipv4.addresses[0].ip }}"
      when: ansible_facts.net_interfaces.Traceback is not defined

```

```

- name: Assert that we learned the tacacs source IPv4 address
  assert:
    that:
      - tacacs_source_ip is defined

- name: Change tacacs server 1 configuration
  cisco.nxos.nxos_aaa_server_host:
    state: present
    server_type: tacacs
    key: "{{ tacacs_key }}"
    address: "{{ tacacs_server_ip1 }}"

- name: Add/Update network device in ISE-1
  ise_network_device:
    ise_node: "{{ ise_node }}"
    ers_user: "{{ ers_user }}"
    ers_pass: "{{ ers_pass }}"
    device: "{{ inventory_hostname }}"
    tacacs_shared_secret: "{{ tacacs_key }}"
    ip_addresses:
      - "{{ tacacs_source_ip }}"
    presence: present

- name: login to device and test aaa on ISE-1 for vrf
  cisco.nxos.nxos_command:
    commands: test aaa server tacacs+ {{ tacacs_server_ip1 }} {{ conf_aaa_vrf_name }} {{ ansible_user }} {{ ansible_password }}
    wait_for: result[0] contains authenticated
    retries: 1
    when: conf_aaa_vrf_name is defined

- name: Login to the device and save the running-config to start
  cisco.nxos.nxos_command:
    commands:
      - "write memory"

```

The actual playbooks used at XYZ were more complex because there are

multiple ways of configuring TACACS, and we also had to automatically detect which one was being used. Furthermore, there were two TACACS servers configured per device, rather than a single one, because loss of connectivity to a device would require an operator to travel—in some cases over 200 km—to connect locally to the device’s console. [Example 5-34](#) shows that, with the correct chaining of Ansible tasks, amazing automation is possible.

Quality of Service Across a Heterogenous Installed Base

Quality of service (QoS) is often a pain to configure. The configuration is highly dependent on the platform as well as the operating system. In addition, there is complexity in translating commands from one platform to another, even within the same vendor; translating across vendors is even more complex.

Customer ABC has a heterogenous installed base with a couple different platforms and two vendors. Due to bandwidth constraints, ABC decided to apply QoS to its network. We were tasked with addressing this in a more than 2000 devices across 100 sites.

Initially, we planned to apply QoS manually. We prepared configurations for each platform and scheduled several maintenance windows to gradually apply them.

The first maintenance window came around, and during the verification steps, traffic flows seemed off. After spending a night troubleshooting, we came to the realization that some of the QoS policies were mismatched. We decided to use Ansible playbooks to configure the devices.

We built an inventory file from a management system that was already in place at ABC. We also built a playbook to verify, upon connecting to a device, whether the facts gathered match that management system’s view. This extra verification step was a needed safeguard because it meant that, without a lot of effort, we can quickly identify inconsistencies that would later cause issues and possible network downtime. We ran this playbook for all devices for which we intended to change the QoS configurations. We did not develop automation for the change on the management system for

incorrect device attributes; we did this manually for the identified devices.

With a consolidated inventory, the QoS workflow had three phases and an optional fourth:

Step 1. Gather information on what was configured on the devices.

Step 2. Push the compiled configurations to the devices.

Step 3. Verify the configurations.

Step 4. (Optional) Roll back the configuration.

In the first step, we gathered information on what was configured on the devices. Using that information, we generated the reverse CLI in order to delete the old QoS configuration remnants from the device. It is paramount to avoid leftover configurations. Furthermore, we merged this information with the CLI configuration we had previously built. The result of this phase was configuration files that would replace the current device QoS configuration with the intended QoS configuration.

The second step was to push the previously compiled configuration from step 1 to the actual devices.

The third step was to verify that the configurations were correctly committed to memory and also run more in-depth tests, such as for QoS queues.

The fourth step was optional and triggered only if the verifications from the third step failed.

The playbooks we developed used conditional sentences to match the configuration to the platform and operating system, along with the Ansible modules to use. They were similar to in the one shown in [Example 5-21](#).

The Ansible solution provided helped us finish the project without any other misconfiguration, and it also provided a tool that the customer can use to provision new network devices without the fear of applying incorrect QoS settings. Furthermore, we continue to use these playbooks in customer deployments today.

Disaster Recovery at a European Bank

A bank in a country in Europe has many data centers. Due to its business

nature, the bank has strict requirements when it comes to availability, reliability, and disaster recovery. The data center technology stack for this bank consisted of a software-defined networking solution, Cisco ACI, and the compute was virtualized using VMware technologies.

Note

Cisco APIC (Application Policy Infrastructure Controller) is the controller for the Application Centric Infrastructure (ACI) software-defined networking solution for data centers.

One of the bank's applications is not distributed due to security concerns, which means it runs on a single data center. The bank engaged us to investigate how to prevent this application from being affected by a failure in that data center.

Several options were considered, but all of them included extending the application to several data centers, such as stretched subnets. These options were not acceptable based on the client's network architecture principles.

However, we found another solution. Instead of providing high availability to this application, we could provide it with disaster recovery. This means that, in the case of a failure, we could start it somewhere else (in another data center). We went a step further and decided to provide disaster recovery for most of the non-redundant applications in that data center. As you might guess, we accomplished this by using Ansible.

We developed some playbooks that fell into two categories:

- Saving the data center state
- Deploying a data center “copy”

Saving the data center state involved playbooks run on the data center where the non-redundant applications live. We decided to run them once at night because the bank's infrastructure does not change often; however, the playbooks can be run as often as needed.

These playbooks capture the state of the network and computing infrastructure—that is, which IP addresses the applications have, in which

VLANs they run, gateway IP addresses, and so on. We extracted this information from the bank’s VMware vCenter and Cisco APIC controller by using Ansible modules.

We then saved the infrastructure state to two databases (for high availability). This allowed us to get a picture of what was running the day before in the data center. A database is not required, however; it would be possible to save the retrieved values to any storage format (for example, text files).

For deploying a data center “copy,” we developed a playbook to read from the database and deploy those concepts. This playbook is run manually in the event of a disaster in the other data center. It copies all the infrastructure that was running there the day before.

What enabled us to do this was not only network automation using Ansible. Importantly, the customer had a software-defined networking data center where all the network infrastructure was represented in a text format. Although the copying we did would have been possible in a traditional data center, connecting to a multitude of devices to gather their state and replicate that on a possibly different topology would have been much more complicated. In addition, vCenter has a central picture of the environment, and that enabled the copy on the computing side.

Using only Ansible, we were able to create a disaster recovery solution with a maximum recovery point objective (RPO) of 24 hours (or less, if the first playbook is run more often) and recovery time objective (RTO) of around 15 minutes.

Plenty of enhancements could be made to this approach. For example, it would be possible to create an automated way of triggering the data center “copy” in the event of an outage. You saw some monitoring tools capable of triggers in [Chapter 3](#). However, it is important to avoid overautomating. You should automate only what brings you benefit and meets your business goals. You will learn more about this in [Chapter 7, “Automation Strategies.”](#)

Summary

This chapter ties the Ansible concepts you learned in [Chapter 5](#) to actual network automation tasks. It goes over interactions with common

components such as the following:

- Files
- Different device types (routers, switches, virtual machines, servers)
- Containers and Kubernetes
- Public cloud components
- APIs

This chapter also presents case studies where Ansible was successfully used to automate processes in projects with thousands of devices. You have seen how Ansible can help you reduce human error, improve agility and speed, and reuse previous work in networking tasks.

Take a moment to think about what configuration, monitoring, or other types of tasks you do regularly. Based on what you have read in this chapter, think about how you would automate them.

Review Questions

You can find answers to these questions in [Appendix A, “Answers to Review Questions.”](#)

1. True or false: When retrieving a remote file’s content, you can use the Ansible *lookup* module.
 - a. True
 - b. False
2. When using the Ansible *slurp* module, what is the output format?
 - a. Base-64
 - b. Binary
 - c. YAML
 - d. JSON

- 3.** You want to copy the result of a task's execution to a file in the local execution machine. You are using the *copy* module, but it is saving the file in the remote host. What module do you use to enhance your solution and save the result locally?
- a.** *slurp*
 - b.** *delegate_to*
 - c.** *when*
 - d.** *msg*
- 4.** You want to retrieve the IP addresses of all interfaces in a Cisco network router. Can you achieve this using only *ansible_facts*?
- a.** Yes
 - b.** No
- 5.** You want to execute several commands on Linux servers from your company after they are physically installed. Which module is the most appropriate?
- a.** *command*
 - b.** *slurp*
 - c.** *ios_config*
 - d.** *yum*
- 6.** You are tasked with installing Ansible in an automated fashion in several hosts. These hosts have different operating systems. Which variable from *ansible_facts* can you use to correctly choose the installation procedure?
- a.** *os_family*
 - b.** *nodename*
 - c.** *ipv4*

d. architecture

- 7.** You must interact with custom software made in-house in your company. There is no Ansible module for it. Do you need to use an automation tool other than Ansible?

 - a.** Yes, you must use a different tool.
 - b.** No, you can still use Ansible.
- 8.** You are tasked with crawling your company website's content using Ansible. Which is the preferred way to achieve this task?

 - a.** Use the *uri* module
 - b.** Use the *command* module
 - c.** Use the *lookup* function
 - d.** Use the *copy* module
- 9.** Your company uses NETCONF as its management protocol. You are tasked with translating some of your previously developed **curl** scripts to Ansible. Which Ansible module would you use to retrieve the configuration?

 - a.** *netconf_config*
 - b.** *netconf_rpc*
 - c.** *restconf_get*
 - d.** *netconf_get*
- 10.** You must delete from your VMware environment hundreds of unused virtual machines that are currently shut down. You decide to use Ansible to automate this task. Which state should you set in the module for this effect?

 - a.** *present*
 - b.** *absent*

c. *shutdownguest*

d. *poweredoff*

Chapter 6. Network DevOps

This chapter covers network DevOps (NetDevOps), a variant of DevOps focused on networking that has grown quite popular lately. This chapter focuses on what NetDevOps is, what it can do for your network, when you should choose this approach, and how to start using it for your use cases.

NetDevOps encompasses practices such as continuous integration (CI), continuous delivery (CD), and infrastructure as code (IaC). This chapter covers these practices and looks at some of the tools (such as Jenkins) you can use to implement them. This chapter also guides you through the process of creating your own mini NetDevOps environment.

NetDevOps uses automation to achieve its goal. Therefore, this chapter touches on many of the technical concepts you've already learned in this book. Although you can use any automation tool, this chapter's examples focus on previously tools you've already used, such as Ansible.

This chapter closes with case studies of real companies that have implemented NetDevOps in their massive-scale networks, highlighting the challenges it has solved for them and how.

What NetDevOps Is

The term *DevOps*, which comes from the software world, brings together development (Dev) and operations (Ops). Before DevOps, developers focused their efforts solely in building working code. After new code was released, the code was deployed to production systems, and operations teams then took care of it. Because these two teams worked in silos, code was often not ready for the production system, which raised many challenges. A famous phrase that shows the lack of collaboration is “It works on my machine!”

DevOps is a development practice and mindset that uses agile principles (collaboration, communication, use of the right tools) to meet the goal of

building better software. It is often illustrated as an infinity symbol with phase names inside it to represent its continuous nature (see [Figure 6-1](#)).

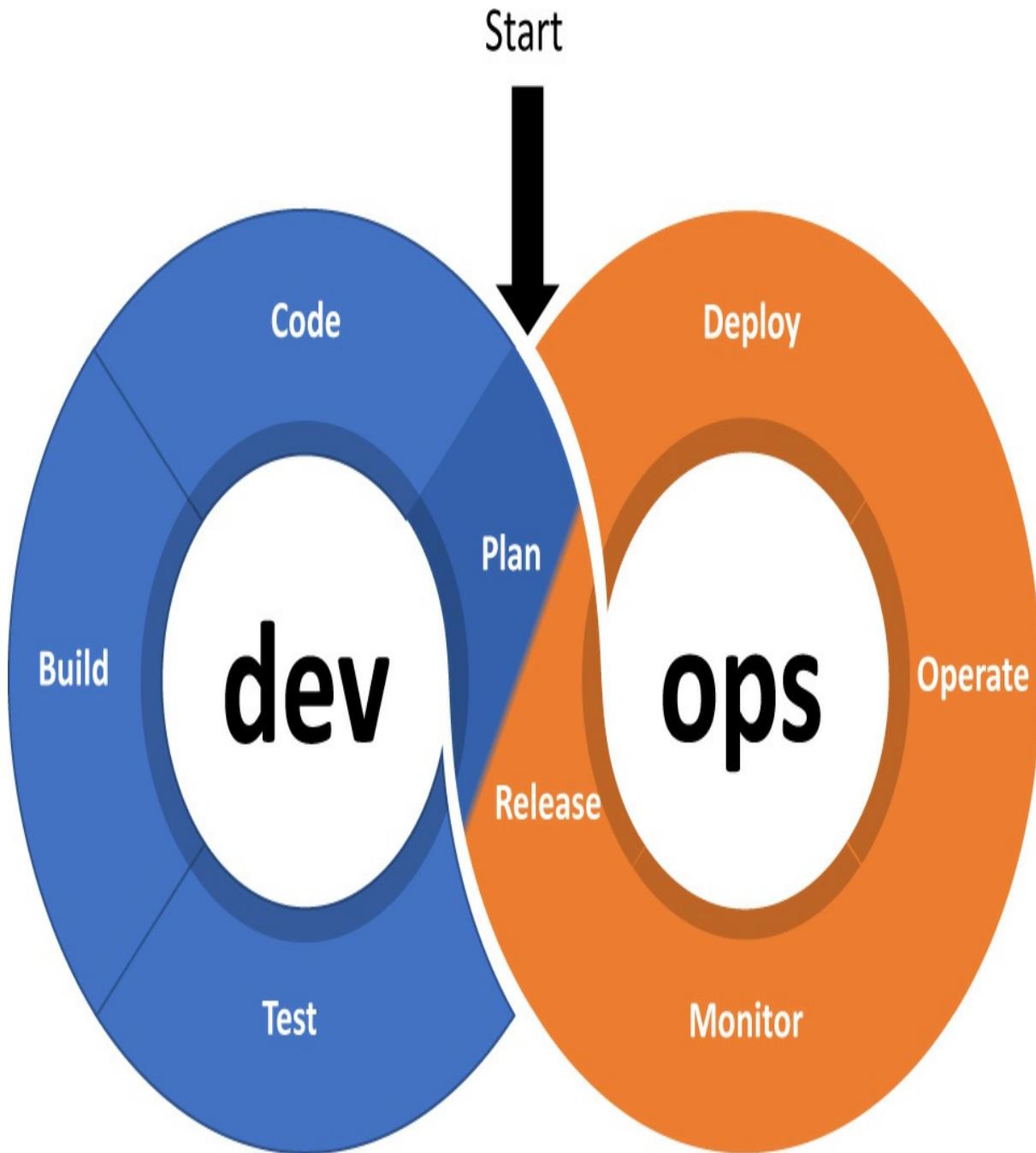


Figure 6-1 DevOps Phases

Starting in the plan phase, teams collect and document requirements to define and describe what they building and what it solves. They also create

metrics to measure and track progress.

In the code and build phases, teams create the code. Multidisciplinary teams, as mentioned previously, collaborate using version control systems to create code and build artifacts that solve what was agreed in the planning phase. (Continuous integration plays a role in these phases.)

During the test phase, code and artifacts are tested to make sure they work as intended, both standalone and integrated with others.

In the release and deploy phases, the artifacts produced are delivered to production systems using automated systems. (Concepts like IaC and continuous development come into play here.)

Finally, in the operate and monitor phases, the newly produced code is in production, where maintenance, monitoring, and troubleshooting occur as necessary.

Why should you care about this software development methodology if you work in networking? Well, it turns out the networking industry has suffered from many of the same issues that plagued the software industry before DevOps. You have probably heard statements like these:

- “We cannot update the network. The business does not allow it.”
- “Every time we implement a network change, something goes wrong.”
- “The network is too critical to be upgraded.”
- “Joe is on vacation, and without him, we cannot make changes to the network.”

As these statements indicate, people perceive networks as being critical but very fragile. They tend to think that with the slightest touch, an outage can occur, and monumental business damages can result.

I don’t know for certain why people have these perceptions, but a fair assumption would be that making network changes sequentially and manually led to these ideas. Many organizations hand out tasks to each engineer involved in a maintenance window and hope for the best. When those tasks don’t go well, rollbacks and outages occur, and even when they do go well, snowflake networks often result, where each switch has a different configuration and nomenclature.

Network changes may take several maintenance windows in sizable networks if the changes are made to individual devices, one by one, by copy and pasting configurations. In networks of hundreds or thousands of devices, a change may take tens of hours.

NetDevOps helps change these types of networking problems by adopting software DevOps practices for network engineering and operations.

NetDevOps could consist of treating the network as code, storing configurations in a source control system like Git, using CI/CD to build and deploy changes to the network, testing before implementing changes, or using APIs in the network—or a combination of all the above.

Traditionally, network configuration changes were made in a workflow-type fashion. NetDevOps replaces that manual workflow with an automated one that operates more quickly and accurately. NetDevOps phases are quite similar to DevOps phases but with a focus on network infrastructure configuration. In the first phase, a team plans and defines what the change should achieve (for example, creating a new VLAN and interface VLAN for some newly acquired servers).

Second, the team gets the current network devices' configurations from a source control repository and makes the required changes. To do so, the team creates configuration templates and substitutes variables for values.

During the test stage, the newly made changes are tested. The testing can apply to just syntax, or it can be extensive end-to-end testing, depending on the critically of the configuration change. The test targets may be virtual or physical devices.

If all tests succeed, during the release and deploy stages, configurations are finally pushed to the production equipment. In a NetDevOps environment, this is no longer a manual change; rather, it is an automated change using, for example, Ansible playbooks.

Finally, the operate and monitor stage involves network monitoring and troubleshooting, as necessary.

[Figure 6-2](#) illustrates these phases of NetDevOps.

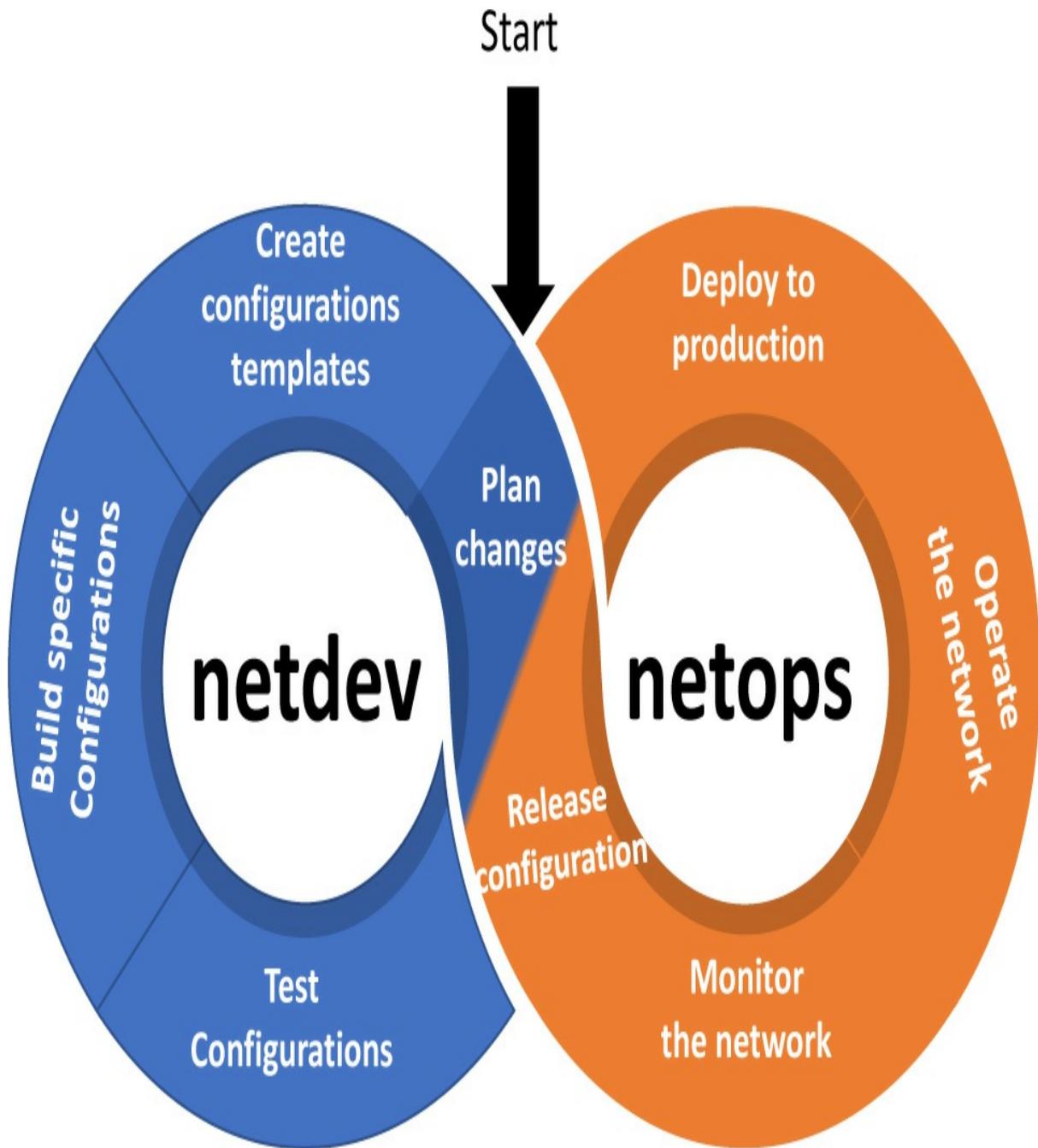


Figure 6-2 NetDevOps Phases

Note

A common misconception is that NetDevOps replaces automation. It does not replace automation. Rather, NetDevOps uses automation to achieve its goals.

DevOps and NetDevOps have, as building blocks, practices that make these phases possible and mostly automated. These processes include the following:

- Source control
- Infrastructure as code (IaC)
- Continuous deployment
- Continuous integration
- Continuous delivery
- Continuous monitoring
- Continuous testing

Source Control

A source control system, sometimes referred to as a *version control* system, allows code developers to collaborate on projects while maintaining accountability and tracking any changes made. Source control is an essential tool for code development, and it is commonly used in the software industry.

The most used and most well-known source control system is Git. We cover it in more depth later in this chapter.

A source control system provides a way of maintaining a single source of truth for projects, even when the development is distributed among different individuals. During a project, developers are constantly writing new code and changing existing code. For example, one developer might be fixing a bug while another is developing a new feature for the same project. These individuals can work in parallel toward a common goal and collaborate using the source control system.

You can imagine a source control system as a storage facility, where users must identify themselves to make changes (deposits or withdrawals), and any change is written to a ledger. If you want to know who made a particular change or what the change was about, you can simply consult the ledger. It is a simple concept but very powerful.

A source control system, regardless of the tool used to implement it, typically provides benefits such as the following:

- **Change tracking:** The system tracks every change made to the code base.
- **Version history:** This system shows the state of every previous version.
- **Accountability:** This system indicates who made changes and when.
- **Authorization:** Only authorized parties can make changes.
- **Traceability:** This system makes it possible to trace and connect changes to comments and other systems.

There are several types of source control systems:

- Local
- Centralized
- Distributed

Local source control systems are rarely used anymore. Such a system consists of a local database that keeps track of changes. For example, [Figure 6-3](#), shows a local system with two versions of the same file. This system does not enable collaboration.

Local machine

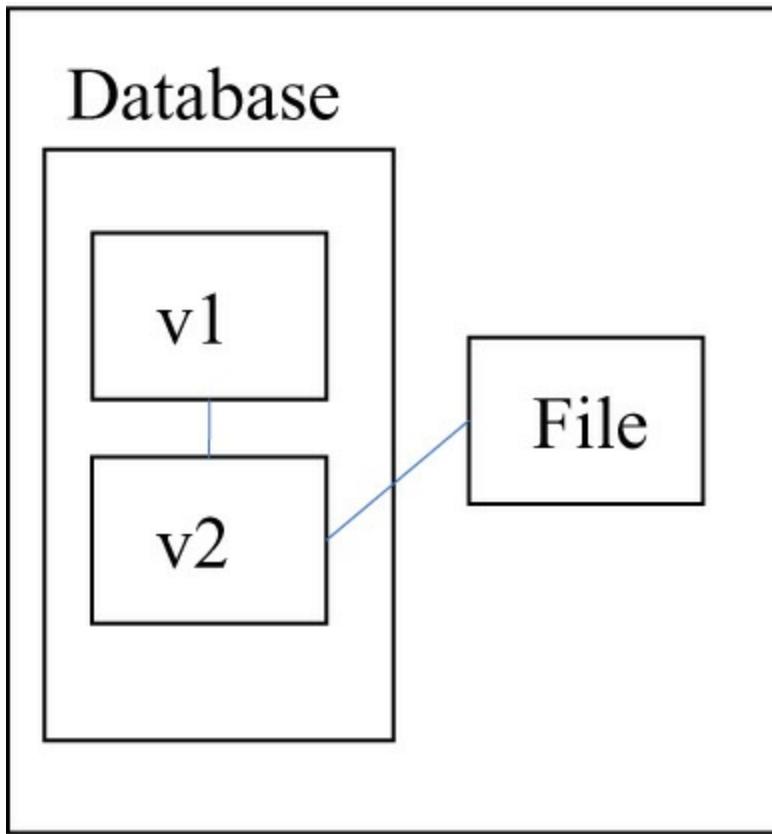
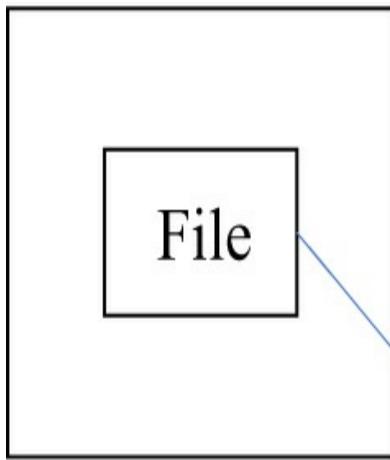


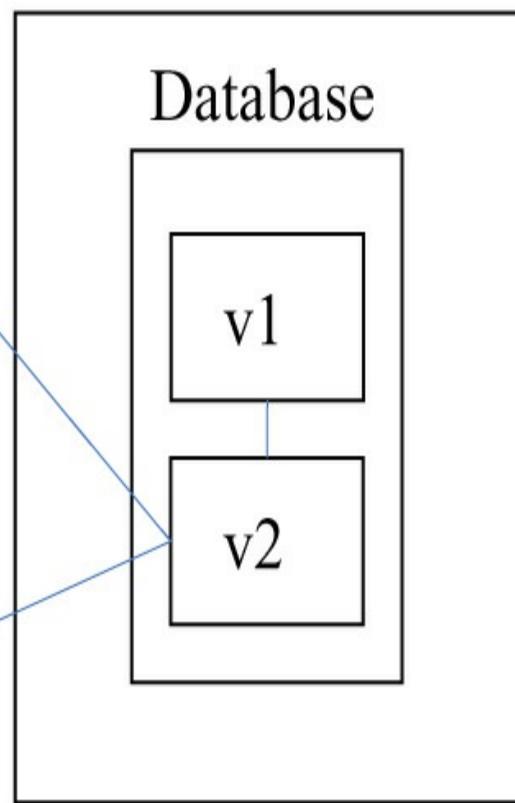
Figure 6-3 Local Source Control

A centralized source control system contains all versioned files, and users interact with this system. It is far superior to a local system, enabling users to know and interact with everyone else. However, as in all centralized systems, there is a big downside: If the system is down, users cannot interact with it; they can't save changes or fetch other users' work. A centralized source control system basically puts all the eggs in one basket (see [Figure 6-4](#)).

Local machine 1



Central Source Control System



Local machine 2

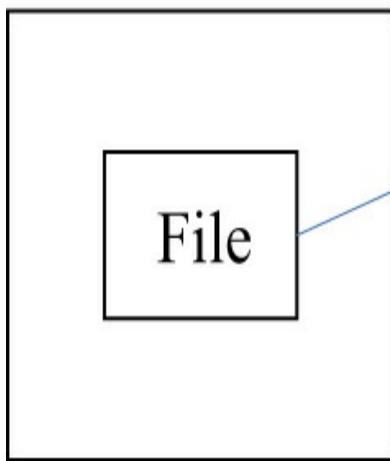
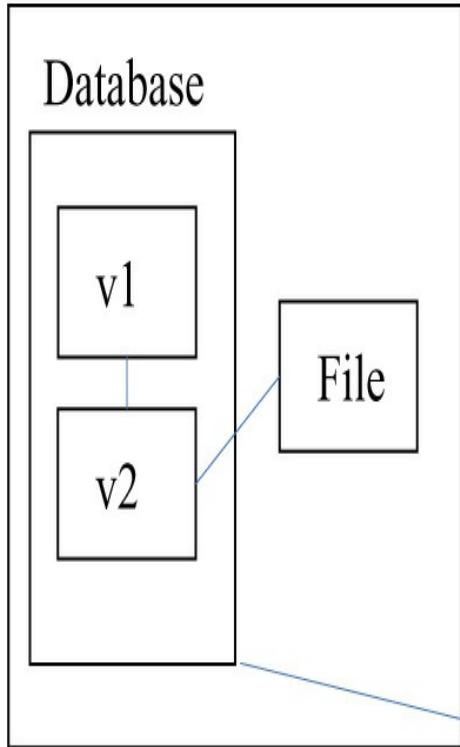


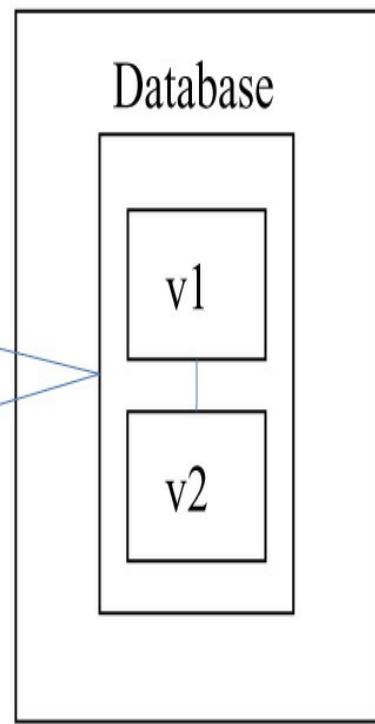
Figure 6-4 Centralized Source Control

A distributed source control system maintains a full copy of the changes in every user's system as well as in a server. For example, [Figure 6-5](#) shows such a system, where each user has a fully copy with all versions. This system addresses the shortcomings of the centralized architecture: If something happens to the central server, you can just replace it with a copy from your local environment. Git uses this architecture.

Local machine 1



Source Control System



Local machine 2

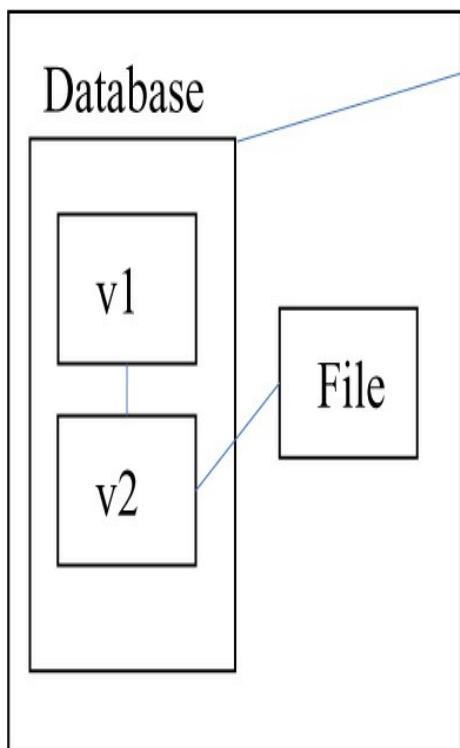


Figure 6-5 Distributed Source Control

With DevOps, developers use source control systems to store code. With NetDevOps, network engineers store network configurations or abstractions of network configurations in source control systems. This makes it possible to know what is configured and where at all times, assuming that changes are not made to the devices without also being made in the source control system. Furthermore, network engineers use these systems to store all kinds of code they use (for example, Ansible playbooks).

Say that you want to change the OSPF hello timer in your network. You have the network configurations for all your routers stored in a version control system, with one text file per router. You alter the hello timer to 5 seconds by committing new versions of these files to the version control system, which triggers an Ansible playbook that deploys the changes to the actual routers. After the change is applied, you observe that the neighborship is correctly reformed. So far it is great, but you go on vacation the following week. During that week, there is a network outage that your colleagues relate to the OSPF timers. Your colleagues can quickly roll back the change by going to the version control system and restoring the previous version of the configurations. You can image that without a version control system in place, this rollback would be much more difficult to perform. First, your colleagues would need to figure out what timers were there before your change, what devices you applied the change to, and so on. Applying any change to a production system requires a deep understanding of the ecosystem, so you can roll back the change, if necessary. With a version control system, it is not necessary to have such a deep understanding because the ecosystem state is versioned.

Hopefully, you can see that change tracking and traceability are game changers in network management. In a NetDevOps scenario, you should store both network device configurations and automation code in a source control system.

Infrastructure as Code (IaC)

Infrastructure as code involves managing and provisioning infrastructure

through configuration definition files instead of using physical hardware configuration or interactive configuration tools. These configuration definition files can be stored and versioned in a source control system. You have seen examples of IaC throughout this book.

Typically, infrastructure was not managed using code; rather, it was managed and provisioned manually. As you have seen, manual management and provisioning are subject to a lot of issues. Infrastructure as code brings several advantages in comparison, including the following:

- Reduction and/or eradication of configuration drift
- Ease of replication
- Repeatability at any scale
- Ease of distribution

With IaC, the state of the infrastructure is represented in a text file. Typically, this file is stored in a central repository. If all changes are made using that file, the resulting infrastructure reflects it. No ad hoc modifications should be made to devices directly; ad hoc modifications lead to snowflake networks and configuration drift.

If you have your configuration in a code format, you can simply change variables to deploy it to another environment or at another scale. If you are using Ansible, and you have a playbook that creates 10 virtual machines in a VMware environment, by simply using a different inventory, you can deploy the same 10 virtual machines to a different vCenter instance.

Note

With IaC, it is important to keep code and configurations separate.

Furthermore, you can share your configuration files with your peers. For example, if you have built a lab for your CCIE training, you can easily share the file that represents the lab with a colleague who is also preparing for the CCIE exam. Your colleague can then re-create your lab topology with minimum effort.

The goal with IaC is to be able to represent all your infrastructure as code,

including network equipment, servers, virtual machines, and load balancers. However, you can also use IaC to represent only a portion of your infrastructure as code.

You need to understand a few key concepts related to IaC:

- Idempotence
- Mutable versus immutable infrastructure
- Imperative and declarative approaches

Idempotence means that a deployment command always sets the target environment to the same configuration, regardless of the environment's starting state. [Example 5-19 in Chapter 5, “Using Ansible for Network Automation,](#)” shows a playbook that installs httpd, but if httpd is already installed it, Ansible does not install it again. This is idempotence: No matter what state the target system is in, Ansible will make sure the playbook's intent is fulfilled (in this case, ensuring that httpd is installed). Idempotence is a default part of some IaC tools but it is not present in all of them. For NetDevOps specifically, idempotence plays an important role as you do not want to end up with conflicting configurations; therefore, accounting for what is already configured is key.

Immutability links to mutable and immutable infrastructure. It represents something that cannot change. An immutable virtual machine is a virtual machine that cannot handle changes. For example, if you have Python 2.7 installed in that virtual machine and you want to perform an upgrade to Python 3.2, you must destroy that virtual machine and re-create another one with the newer version of Python. There are advantages and disadvantages to this approach. Some advantages are improved security posture and reduced operational complexity because a configuration system does not have to calculate the delta between where the system is and where the new configuration wants it to be. However, re-creating infrastructure every time can mean added cost. You can choose whether to use IaC with mutable or immutable infrastructure. For NetDevOps specifically, immutability plays a lesser role because most of the devices are physical and therefore must be reconfigured. However, if you are managing computing resources, such as virtual machines, immutability may be more important.

With IaC tools, another choice is declarative or imperative approaches. The

key difference between them is that with the declarative approach, you only define what you want, whereas with the imperative approach, you specify how you will get it.

With the declarative approach, you specify the final desired state, and the tool handles every step in between where you are and where you want to go. This is neat, as it means you offload the responsibility of figuring out the order in which tasks should be executed to the tool. Terraform is an example of a tool that provides a declarative approach.

The imperative approach, on the other hand, is a procedural approach. You must specify step by step what you want done and in which order. Ansible is an example of a tool that takes an imperative approach. (Although there are Ansible modules that are declarative, they are rare.) The imperative approach is typically easier for network administrators as it's similar to their traditional manual workflows.

A way of visualizing the difference between these approaches is to look at the process of making changes. Say that you want to create two virtual machines in AWS using Ansible. You can do so by using the Ansible module `ec2`, as shown in [Example 6-1](#).

Example 6-1 Deploying Two VMs by Using the `ec2` Module in Ansible

```
---
- hosts: all

  tasks:
    - amazon.aws.ec2:
        count: 2
        instance_type: t2.micro
        image: "ami-08d9a394ac1c2994c"
```

If you later want to have three VMs instead of the initial two, you would need to provision one more. To do that, you could run the playbook in [Example 6-2](#).

Example 6-2 Deploying Another VM by Using the `ec2` Module in Ansible

```

---
- hosts: all

  tasks:
    - amazon.aws.ec2:
        count: 1
        instance_type: t2.micro
        image: "ami-08d9a394ac1c2994c"

```

As you can see in this example, you have to figure on your own the difference between the final wanted state (three VMs) and the number of virtual machines already deployed (two VMs), and you use that information to run an edited version of the playbook. You could have the same playbook and simply receive the count as an input variable; however, that would not change the fact that you must know what variable to pass.

Let's look at the same scenario but now with a declarative tool. With Terraform, you would start by defining the two VMs you need initially (see [Example 6-3](#)).

Example 6-3 Deploying Two VMs with Terraform

```

resource "aws_instance" "ec2name" {
  ami           = "ami-08d9a394ac1c2994c"
  instance_type = "t2.micro"
  count         = 2
}

```

Later, you need three virtual machines; with a declarative approach, you simply need to alter the count to three, as in [Example 6-4](#). Terraform figures out that out of the three desired virtual machines, two already exist; it therefore deploys only one more.

Example 6-4 Deploying Another VM with Terraform

```

resource "aws_instance" "ec2name" {
  ami           = "ami-08d9a394ac1c2994c"
}

```

```
instance_type = "t2.micro"
count         = 3
}
```

Note

Examples 6-1 to 6-4 are simplified and do not show the complete code.

To summarize, in a NetDevOps scenario, you represent your network as code. There are many different ways to do this, including by using abstractions or entire configurations. You will see examples of both later in this chapter.

Continuous Integration and Continuous Deployment/Delivery (CI/CD)

CI/CD is an acronym that is widely used in DevOps. CI stands for continuous integration, and CD can stand for continuous delivery or continuous deployment.

Continuous integration is a software practice that involves integrating the developed code as much as possible, with a big emphasis on testing and increasing the frequency of deliveries. With CI, developers integrate their code in a shared repository, and that code is verified by automated builds and tests. Continuous integration emphasizes use of automation to make this process quick and reliable.

A key benefit of CI is that it makes it possible to detect errors quickly and early in the development process, which drives down development costs. Remember that frequent integrations make code changes small.

The following are some key building blocks of CI that are worth highlighting:

- Revision control
- Build automation

- Automated testing

With NetDevOps, you may choose to make small but frequent changes in your network, keep the configuration in a revision control system, and perform automated testing. Testing requires a test network (which can be virtual), and it involves submitting the changes you want to make for integration testing with a replica of the production environment. This requires a shift in mindset because it is common in networking to apply changes directly to the production environment.

Continuous deployment, which is an extension of continuous integration, is a software release process. If the code build passes all the tests, this build is pushed to the production environment.

Continuous delivery is a step back from continuous deployment. Even if a build passes all the integration tests, human intervention is required before this build is pushed to production. It provides a safety measure.

For NetDevOps, CD involves pushing tested configurations to the production environment automatically (by using Ansible, for example).

Deciding between continuous deployment and continuous delivery is highly dependent on the network environment in question. The choice comes down to agility (continuous deployment) versus control (continuous delivery).

CI/CD decreases the time it takes to get a solution in production, which, in turn, decreases the time to market and increases the organization's agility in responding to customer requests.

The benefits of CI/CD for software can be summarized as follows:

- Allows earlier and easier detection of defects
- Reduces testing efforts
- Reduces time to deliver new software
- Provides a quicker feedback mechanism

For NetDevOps, most of these benefits as well, but with a networking slant:

- Allows earlier and easier detection of configuration defects
- Reduces testing efforts
- Reduces time to deliver network features

- Provides a quicker feedback mechanism for network changes

CI/CD together are implemented using CI/CD servers. We cover some options for these servers later in this chapter, and we also examine the characteristics to consider when choosing such a server.

In the CI/CD servers, you create your workflows, often called *pipelines*. It is possible to automate these workflows. For example, say that you want to reconfigure the SNMP community in five network devices. Your workflow might involve the following steps:

Step 1. Connect to a device by using SSH.

Step 2. Retrieve the current SNMP community configuration.

Step 3. Delete the current SNMP community configuration (typically with the **no** keyword).

Step 4. Add the new SNMP community configuration.

Step 5. Try to retrieve an SNMP OID by using the newly configured community.

Step 6. Repeat steps 1 through 5 for the remaining four devices.

In a CI/CD server, such as Jenkins, each of these steps would be a stage. And inside those stages, you could use automation tools such as Ansible to make configuration changes. You could also verify the configuration by using Ansible or a Bash script. After building a pipeline like this, you could rerun it any time you need to change the SNMP community to a different one or in a different set of devices. You could also allow it to be externally triggered (for example, by an API call).

There are costs to adopting CI/CD, but most of the time the benefits outweigh them.

Why Use NetDevOps

NetDevOps is increasingly being adopted by companies as their networks grow in size and complexity. If you find yourself struggling with any of the following, NetDevOps can probably help you:

- Different configurations in different device (that is, snowflake networks)
- Network changes taking too long
- Lack of a single source of truth about network devices' configurations
- The network being the bottleneck in your business when it comes to releasing a new product/service
- Network changes leading to service outages
- Lack of accountability for configuration changes
- Your network being functional but fragile

NetDevOps aims to control infrastructure by using a consistent version-controlled method that deploys changes in a parallel and automated manner. It enables you to do the following:

- Keep all configurations consistent
- Test changes before executing them
- Automate changes
- Maintain a history log
- Perform small and regular changes

You might not want to do all of these things. For example, you might not want to test your changes (although you definitely should) because you do not have a test network available. You can tailor NetDevOps, to your needs.

When to Use NetDevOps

After reading about the benefits of NetDevOps, you probably want to jump right in and implement it. However, it is hard to implement NetDevOps straight into a legacy network. The best idea is to start small and grow.

NetDevOps is much easier to implement in a cloud (either public or private) or in a software-defined networking (SDN) solution than in a legacy network. In cloud and SDN solutions, you typically have a single controller that manages all configurations; in contrast, in legacy networks, each component has its own configuration. This difference directly influences whether you

can treat your network as code. [Example 6-5](#) shows an Ansible playbook that deploys a new subnet on AWS.

Example 6-5 Deploying a New Subnet in the Cloud with Ansible

```
$ cat subnets_playbook.yml
---
- hosts: localhost
  gather_facts: False

  tasks:
    - name: import vars
      include_vars: subnets.yml

    - name: create ec2 vpc subnet
      ec2_vpc_subnet:
        vpc_id: "vpc-d7e955bd"
        region: "eu-central-1"
        az: "eu-central-1a"
        state: present
        cidr: "{{ item }}"
      with_items:
        - "{{ subnets }}"

$ cat subnets.yml
subnets:
  - "172.31.11.0/24"
  - "172.31.10.0/24"
```

To add a new subnet, you could simply add another line to the subnets.yml file and push it to your Git repository. If you had a trigger for this action, a pipeline would run the playbook, and the new subnet would be deployed.

Legacy network devices are different. [Example 6-6](#) shows an example of a playbook to deploy a new subnet on a topology with a single access switch and a distribution switch. This example verifies the role of the switch based on its inventory hostname and deploys a specific template. For example, this playbook creates the SVI on the distribution switch and only creates the

Layer 2 VLAN on the access switch.

Example 6-6 Deploying a New Subnet on a Legacy Network with Ansible

```
$ cat vlan_deployment.yml
---
- name: Ansible Playbook to configure a vlan
  hosts: ios

  tasks:
    - name: import vars
      include_vars: subnets.yml

    - name: render a Jinja2 template onto an IOS device
      cisco.ios.ios_config:
        src: "{{ dist_template.j2 }}"
        when: "'dist_switch01' == '{{ inventory_hostname }}'"

    - name: render a Jinja2 template onto an IOS device
      cisco.ios.ios_config:
        src: "{{ access_template.j2 }}"
        when: "'access_switch01' == '{{ inventory_hostname }}'"


$ cat access_template.j2
{% for item in vlan %}

vlan {{ item.id }}
  name {{ item.name }}

{% endfor %}
$ cat dist_template.j2
{% for item in vlan %}

vlan {{ item.id }}
  name {{ item.name }}

interface vlan {{ item.id }}
description Ansible
```

```
ip address {{ item.subnet }}  
  
{% endfor %}
```

To trigger this playbook, you could make a modification on the subnets.yml file with a Git trigger. [Example 6-7](#) shows a file being used in the configuration templates from the previous example, with Jinja2 accomplishing the substitution.

Example 6-7 Using a Variable File to Add Subnets

```
$ cat subnets.yml  
vlans:  
  - id: 101  
    name: servers  
    subnet: 172.31.11.0/24  
  - id: 202  
    name: cameras  
    subnet: 172.31.10.0./24
```

NetDevOps is based on network as code. In legacy networks, representing your network as code is harder to do. Creating good abstractions for network as code is a very important step to accurately represent your network in the best possible way.

One way of implementing NetDevOps in a legacy network is by storing all switch/router configurations in a source control system; when changes are made to those configurations, you test them, either in a virtual environment or in a physical staging environment, and push them down to the real devices. (You will see an example of this in the next section.) Storing whole configurations instead of abstractions has drawbacks in terms of management complexity. Nonetheless, it can be good starting point.

Considering the following questions can help you determine whether you are ready for NetDevOps:

- Am I already using network automation (for example, Ansible, Terraform)?

- Are my network configurations stored somewhere?
- Is my network homogenous? (Or does every device have a potentially different configuration?)
- Do I have a testing/staging environment?

Take some time to answer these questions because jumping right in to NetDevOps is not necessarily the best option. If you can answer “yes” for each of these questions, you are ready to embark on a NetDevOps journey.

NetDevOps uses automation to achieve its goals. In previous chapters, you have seen how to automate tasks by using Ansible. If you are already doing this, orchestrating the steps with a CI/CD platform could be your next step. However, if you are not already using Ansible for automation, you should first look into automation.

You need to keep a log trail. Are you storing your configurations somewhere? Or is every switch potentially differently configured, and do you have a hard time knowing when something changes? Addressing these issues would be a great place to start.

Another very important aspect is a test environment. The software industry would not work if changes could not be tested before being deployed to production. In networking, it is a common practice to use a test environment. In order to harvest proper benefits from NetDevOps, you should consider having separated environments.

Note

If you have multiple environment—such as test and production environments—try to keep them as similar as possible. If their configurations drift apart, any testing done in the test environment might not reflect what will happen in the production environment.

If you are not yet there, do not panic. The next chapter covers automation strategies, and you can draft your own strategy to reach a state where you can benefit from NetDevOps.

NetDevOps Tools

DevOps and NetDevOps focus on using the right tools. This section examines two of the most commonly used CI/CD servers: Jenkins and GitLab CI/CD. Jenkins a seasoned veteran in the CI/CD world, and GitLab CI/CD is a newer player.

Many CI/CD tools are available. Choosing the best one typically comes down to the following:

- What you are most familiar with
- Required integrations, if any
- The support model from your vendors
- Tools being used in other departments
- Whether a hosted or nonhosted situation is more applicable

Let's look first at Git, which is the most used distributed version control system and a key piece of NetDevOps.

Git

As mentioned earlier in this chapter, Git is a distributed source control system that tracks changes on project files over time. The process required to install Git depends on the operating system, but it is as simple as installing any other application.

The first thing you do after installing Git is set up your name and email address so Git can identify the changes you make to projects. You can do this by using the **git config** command.

A Git repository is a container for a project tracked by Git. On your machine, you have folders with files, and you can enable Git to track them and turn them into a repository. There are local and remote repositories:

- A local repository is an isolated repository stored in your local machine where you can work on your local version of a project.
- A remote repository is copy of the repository in a remote server.

Remote repositories, which are typically used in collaboration environments, are usually hosted by hosting services such as GitHub or GitLab.

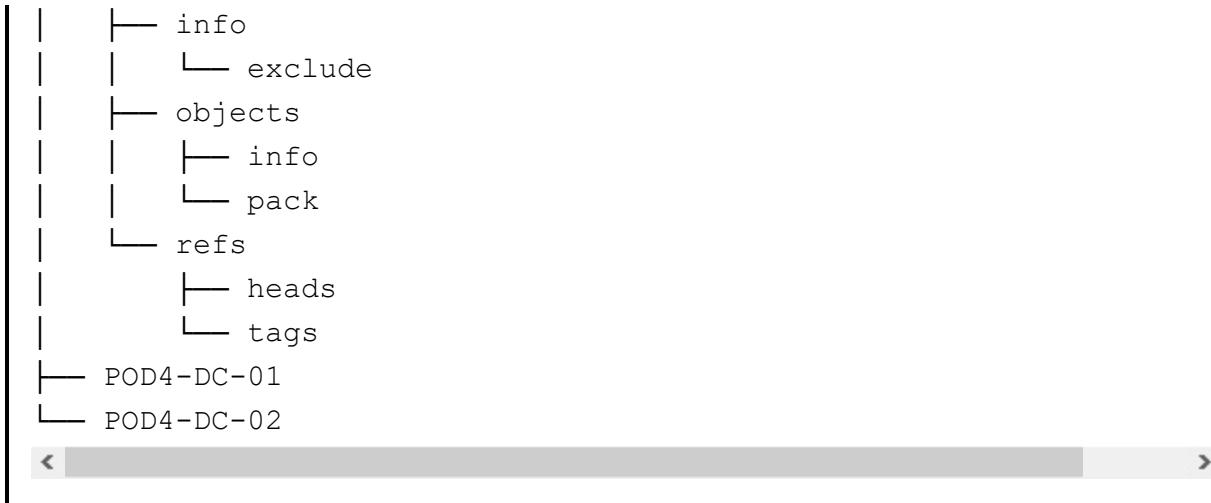
In order to turn a folder into a project, you must issue the **git init** command. This command creates a hidden .git directory, which is what Git uses to track the repository. You can see this in action in [Example 6-8](#). This example starts with a directory with only two text files (POD4-DC-01 and POD4-DC-02) and, when the repository is initialized, a whole hidden directory is added.

Example 6-8 Initializing a Git Repository

```
$ tree
.
├── POD4-DC-01
└── POD4-DC-02

$ git init
Initialized empty Git repository in /Users/ivpinto/Desktop/gitdemo

$ tree -la
.
└── .git
    ├── HEAD
    ├── config
    ├── description
    └── hooks
        ├── applypatch-msg.sample
        ├── commit-msg.sample
        ├── fsmonitor-watchman.sample
        ├── post-update.sample
        ├── pre-applypatch.sample
        ├── pre-commit.sample
        ├── pre-merge-commit.sample
        ├── pre-push.sample
        ├── pre-rebase.sample
        ├── pre-receive.sample
        ├── prepare-commit-msg.sample
        └── update.sample
```



The repository is empty because you have only initialized it. Your configuration files (POD4-DC-01 and POD4-DC-02) are not even being tracked by Git. In order for Git to track changes, you must commit them. With Git, the term *commit* refers to saving the current state. Typically, commits are logical points in a project where you save checkpoints. You can later come back to a checkpoint, sort of the way you can with a rollback, if needed. In [Figure 6-6](#), each of the circles represents a commit in a timeline, and you can see that commits are associated with different messages and hashes.

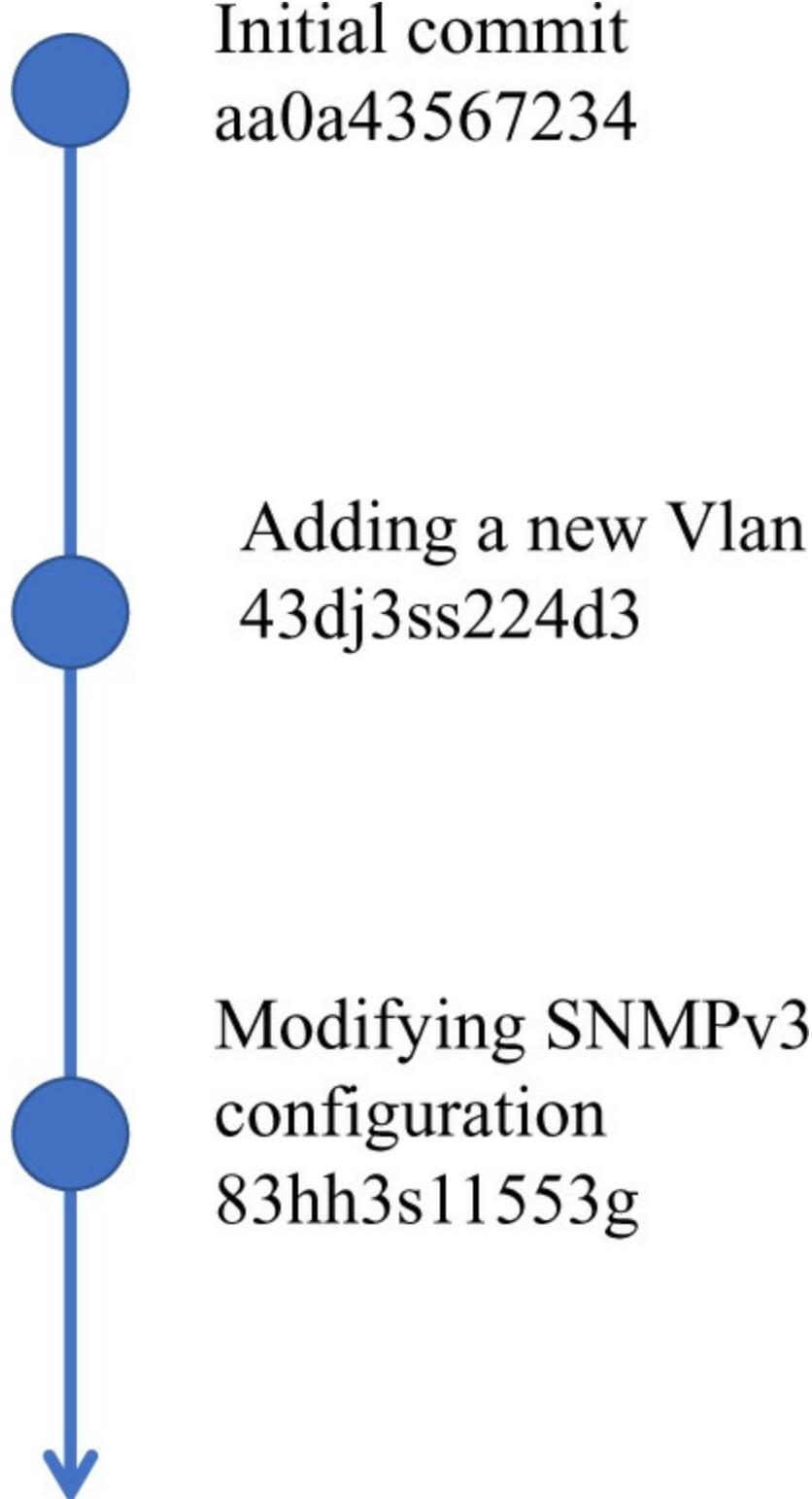


Figure 6-6 *Git Commit as a Checkpoint*

Before you can create a commit, you must stage your changes. Staging

changes involves selecting which of the changes you have made on a project you want to commit. This might sound complicated, but it really isn't.

[Example 6-9](#) expands on the [Example 6-8](#) by verifying which files are being tracked by Git. You can see that Git is not tracking either of the two configuration files. This example also shows the configuration file POD4-DC-01 being added to the staging area with the **git add** command and these changes being committed with the **git commit** command. Finally, you can see in the output of the **git log** command that the commit was successful.

Example 6-9 Adding a File to a Git Repository

```
$ git status
On branch main

No commits yet

Untracked files:
(use "git add <file>..." to include in what will be committed)
    POD4-DC-01
    POD4-DC-02

nothing added to commit but untracked files present (use "git add"
$ git add POD4-DC-01
$ git status
On branch main

No commits yet

Changes to be committed:
(use "git rm --cached <file>..." to unstage)
    new file:   POD4-DC-01

Untracked files:
(use "git add <file>..." to include in what will be committed)
    POD4-DC-02

$ git commit -m "Adding POD4-DC-01 Configurations"
[main (root-commit) 275cc12] Adding POD4-DC-01 Configurations
```

```
1 file changed, 8 insertions(+)
create mode 100644 POD4-DC-01
$ git log
commit 275cc129eef9c780dd3d80a323b4d852cf25a16 (HEAD -> main)
Author: ivpinto <ivpinto@cisco.com>
Date:   Wed May 19 17:50:32 2021 +0100
```

Adding POD4-DC-01 Configurations

If you want to dive deeper into what changes were made, you can use the **git show** command with the commit hash, as shown in [Example 6-10](#). In this case, the commit hash is from the previous **git log** output, and because this is the first time the file has been added, all the content shows the plus sign to mark addition.

Example 6-10 Verifying Changes to a File by Using Git

```
$ git show 275cc129eef9c780dd3d80a323b4d852cf25a16
commit 275cc129eef9c780dd3d80a323b4d852cf25a16 (HEAD -> main)
Author: ivpinto <ivpinto@cisco.com>
Date:   Wed May 19 17:50:32 2021 +0100

        Adding POD4-DC-01 Configurations

diff --git a/POD4-DC-01 b/POD4-DC-01
new file mode 100644
index 000000..454a1fd
--- /dev/null
+++ b/POD4-DC-01
@@ -0,0 +1,8 @@
+hostname POD4-DC-01
+
+tacacs-server host 10.10.0.2
+tacacs-server host 10.10.0.3
+tacacs-server directed-request
+tacacs-server key 7 08020D5D0A49544541
+!
```

```
+ntp server vrf Mgmt-vrf 10.10.10.2
```

One of the benefits of using Git is that you can go back in time, if needed. Say that you make a modification to a file and commit it. However, you discover that the modification is not what you wanted, and you need to go back to a known good state. You can do so by using the **git checkout** command with the commit hash. [Example 6-11](#) shows this workflow. It starts from the state in [Example 6-10](#) and modifies POD4-DC-01 tacacs-server hosts from the 10.10.0.0/24 subnet to the 10.10.1.0/24 subnet and then goes back to an earlier version. This is a fairly quick and reliable rollback technique.

Example 6-11 Rolling Back Changes in Git

```
$ cat POD4-DC-01
hostname POD4-DC-01
tacacs-server host 10.10.1.2
tacacs-server host 10.10.1.3
tacacs-server directed-request
tacacs-server key 7 08020D5D0A49544541
!
ntp server vrf Mgmt-vrf 10.10.10.2

$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working direc
    modified:   POD4-DC-01

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    POD4-DC-02

no changes added to commit (use "git add" and/or "git commit -a")
$ git add POD4-DC-01
$ git commit -m "Changed to new TACACS"
```

```
[main 22e0181] Changed to new TACACS
  1 file changed, 2 insertions(+), 2 deletions(-)
$ git log
commit 22e01810470f50c57fe79a305469e90ae5e64a35 (HEAD -> main)
Author: ivpinto <ivpinto@cisco.com>
Date:   Wed May 19 18:00:45 2021 +0100
```

Changed to new TACACS

```
commit 275cc129eef9c780dd3d80a323b4d852cf25a16
Author: ivpinto <ivpinto@cisco.com>
Date:   Wed May 19 17:50:32 2021 +0100
```

Adding POD4-DC-01 Configurations

```
$ git checkout 275cc129eef9c780dd3d80a323b4d852cf25a16
$ cat POD4-DC-01
hostname POD4-DC-01
tacacs-server host 10.10.0.2
tacacs-server host 10.10.0.3
tacacs-server directed-request
tacacs-server key 7 08020D5D0A49544541
!
ntp server vrf Mgmt-vrf 10.10.10.2
```



Yet another neat functionality of Git is branches. A branch is an individual timeline for commits. So far, all the examples have shown the main branch. By default, a Git repository comes a main branch. When you create a branch, you create a copy of where you are in terms of commits in another parallel timeline. [Figure 6-7](#) illustrates the branch functionality.

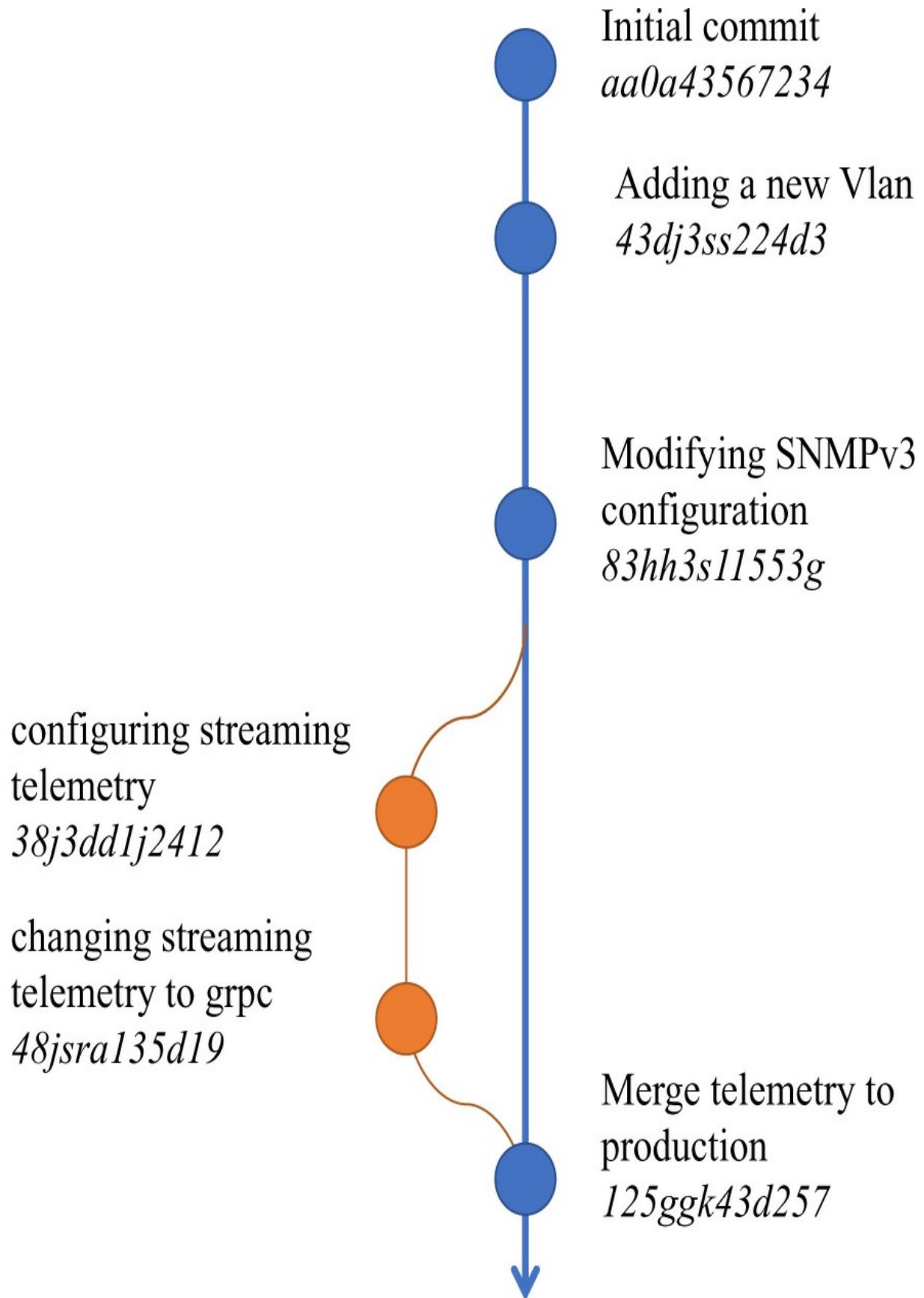


Figure 6-7 Visual Representation of Git Branches

[Example 6-12](#) creates a new branch called *new_device* with the **git branch** command. It adds to this branch a new configuration file for a device called POD4-DC-02 and commits it. You can see from the log that this new commit only exists in the newly created branch and not in the main branch.

Example 6-12 Making Changes in a Git Branch

```
$ git branch
* main
$ git branch new_device
$ git checkout new_device
Switched to branch 'new_device'
$ git branch
  main
* new_device
$ git status
On branch new_device
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    POD4-DC-02

nothing added to commit but untracked files present (use "git add"
$ git add POD4-DC-02
$ git commit -m "Adding new DC device"
[new_device c9cacfd] Adding new DC device
  1 file changed, 8 insertions(+)
   create mode 100644 POD4-DC-02
$ git log
commit c9cacfd2d21861bbef8cc93c6c835d2001fb75c0 (HEAD -> new_device)
Author: ivpinto <ivpinto@cisco.com>
Date:   Wed May 19 18:45:48 2021 +0100

  Adding new DC device

commit 22e01810470f50c57fe79a305469e90ae5e64a35 (main)
```

```
Author: ivpinto <ivpinto@cisco.com>
Date:   Wed May 19 18:00:45 2021 +0100
```

Changed to new TACACS

```
commit 275cc129eef9c780dd3d80a323b4d852cf25a16
Author: ivpinto <ivpinto@cisco.com>
Date:   Wed May 19 17:50:32 2021 +0100
```

Adding POD4-DC-01 Configurations

```
$ git checkout main
Switched to branch 'main'
$ git log
commit 22e01810470f50c57fe79a305469e90ae5e64a35 (HEAD -> main)
Author: ivpinto <ivpinto@cisco.com>
Date:   Wed May 19 18:00:45 2021 +0100
```

Changed to new TACACS

```
commit 275cc129eef9c780dd3d80a323b4d852cf25a16
Author: ivpinto <ivpinto@cisco.com>
Date:   Wed May 19 17:50:32 2021 +0100
```

Adding POD4-DC-01 Configurations

Branches are paramount when developing new features or testing ideas as you do not influence the main branch. For NetDevOps, you should make your changes in different branches. When you finish working in a branch, you typically need to join the work done to the main branch. You can achieve this by using the **git merge** command. There are a variety of merging strategies. For example, you can keep all the history of commits done in a branch, or you can aggregate all the commits under a single commit in the destination branch.

[Example 6-13](#) merges the newly created *new_device* branch to the main branch, and then the device configurations for POD4-DC-02 exist in the main branch.

Example 6-13 Merging a Branch with the main Branch in Git

```
$ git merge new_device
Updating 22e0181..c9cacfd
Fast-forward
  POD4-DC-02 | 8 ++++++++
  1 file changed, 8 insertions(+)
  create mode 100644 POD4-DC-02
$ git log
commit c9cacfd2d21861bbef8cc93c6c835d2001fb75c0 (HEAD -> main, new)
Author: ivpinto <ivpinto@cisco.com>
Date:   Wed May 19 18:45:48 2021 +0100

    Adding new DC device

commit 22e01810470f50c57fe79a305469e90ae5e64a35
Author: ivpinto <ivpinto@cisco.com>
Date:   Wed May 19 18:00:45 2021 +0100

    Changed to new TACACS

commit 275cc129eef9c780dd3d80a323b4d852cf25a16
Author: ivpinto <ivpinto@cisco.com>
Date:   Wed May 19 17:50:32 2021 +0100

    Adding POD4-DC-01 Configurations
```

You have already learned about the most commonly used Git features. However, you have only worked in the local environment, and you know that you can also work with remote repositories. Without a remote repository, it is hard to collaborate with others, as you can take advantage of only a subset of the benefits of Git. You should always set up a remote repository.

There are plenty of ways to implement remote repositories, and here we focus on the popular platform GitHub. However, there are many SaaS and hosted offerings as well, and all the **git** commands work equally well with them.

You need to register an account with GitHub (or another providers). After you create this such, you are in one of two scenarios:

- You have a local repository you want to share.
- There is a remote repository you want to consume.

To share a local repository, all you need to do is to inform Git that there is a remote server. You can do this with the **git remote add** command. Then you can use the **git push** command to push changes to the remote repository and the **git pull** command to retrieve any changes made by your collaborators.

[Example 6-14](#) continues from [Example 6-13](#), adding a remote GitHub server and pushing the local copy to it. The URL used for the remote server is retrieved from the remote repository provider.

Example 6-14 Adding a Remote Repository and Pushing Changes

```
$ git remote add origin https://github.com/IvoP1/gitdemo.git
$ git push -u origin main
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 8 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (9/9), 872 bytes | 872.00 KiB/s, done.
Total 9 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), done.
To https://github.com/IvoP1/gitdemo.git
 * [new branch]      main -> main
Branch main set up to track remote branch 'main' from 'origin'.
```

When you want to consume a remote repository, you can use the **git clone** command followed by the remote URL. [Example 6-15](#) clones the previously created repository to another folder. The remote URL is automatically populated.

Note

When collaborating with other people, you need to set up

permissions on your repositories.

Example 6-15 Adding a Remote Repository and Pushing Changes

```
$ git clone https://github.com/IvoP1/gitdemo.git
Cloning into gitdemo...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 9 (delta 2), reused 9 (delta 2), pack-reused 0
Unpacking objects: 100% (9/9), 852 bytes | 106.00 KiB/s, done.
```

Note

When using GitHub or other provider, you have access to graphical interfaces where you can see changes, perform diffs, and do most of the Git actions.

GitLab CI/CD

GitLab is a source control repository, and GitLab CI/CD is a tool that is built into GitLab for software development. GitLab and GitLab CI/CD are quite intertwined, and their integration is quite straightforward and simple.

GitLab CI/CD is an open-source, free, and self-hosted tool with a well-documented and simple installation. It can integrate with a number of tools, such as Kubernetes and Slack. The configuration files for GitLab CI/CD are written in YAML, which makes them fairly easy to understand.

The GitLab CI/CD tool is embedded directly in your source control repository, and a typical flow goes through these steps:

Step 1. An operator pushes new network device configurations to a branch.

Step 2. A pipeline pushes the configurations to a virtual test network, such as Cisco Cloud Modeling Labs or GNS3.

Step 3. The same pipeline runs tests on those devices (for example, `show` commands).

Step 4. If the test results are acceptable, the configurations can be merged into the main branch.

Step 5. A different pipeline runs on the main branch configurations and pushes the configurations to the production network.

Step 6. If the previous result is acceptable, the pipeline finishes. If not, a rollback is triggered.

This is a simple example, and workflows can be more complex and can include different variables and triggers.

In GitLab CI/CD, pipelines are the top-level construct. The flow just shown would consist of two pipelines. A pipeline has jobs and stages:

- **Jobs:** Define what to do
- **Stages:** Define when to do it

Jobs are executed by runners, which are agents that pick up jobs, execute them, and return results. There are several types of runners, but discussing them is beyond the scope of this book.

Typically, pipelines are triggered automatically (for example, by a code commit), and you can also trigger them manually if required.

A pipeline typically has three stages:

Stage 1. Build stage

Stage 2. Test stage

Stage 3. Production stage

Each stage has jobs defined. For example, the test stage could comprise three test jobs, named test-1, test-2, and test-3. [Example 6-16](#) shows a configuration file that defines a pipeline.

Example 6-16 *GitLab CI/CD Pipeline Definition*

```
build-job:
```

```
stage: build
script:
  - echo "Building stuff"

test-1:
  stage: test
  script:
    - echo "Testing"

test-2:
  stage: test
  script:
    - echo "More testing"

test-3:
  stage: test
  script:
    - echo "Yet more testing"

deploy-job:
  stage: production
  script:
    - echo "Push to production"
```

You would need to modify the script shown in this example and insert actual test and build instructions. Currently, this definition just prints messages with the **echo** command.

GitLab CI/CD is easy to use, and you can see all the constructs described here in its graphical interface.

Jenkins

Jenkins is a CI/CD tool built using Java. It is open source, free, and self-contained. You can easily install Jenkins in a container, on a VM, or natively on a machine. There are also SaaS offerings for Jenkins.

Jenkins functions using a controller/agent architecture. The Jenkins controller

is responsible for the following:

- Scheduling build jobs
- Dispatching builds to the agents for execution
- Monitoring the agents
- Saving and showing the build results
- Executing build jobs directly (only if required)

Jenkins agents are responsible for executing jobs dispatched by the controller.

You don't necessarily need to have agents. In a small environment, a single controller can do every function. However, in big, distributed environments, several agents are typically used due to the need for testing in different environments (for example, Red Hat or Debian). You can even dedicate specific agents to certain tasks. [Figure 6-8](#) illustrates a distributed Jenkins architecture where the agents execute jobs sent by the controller.

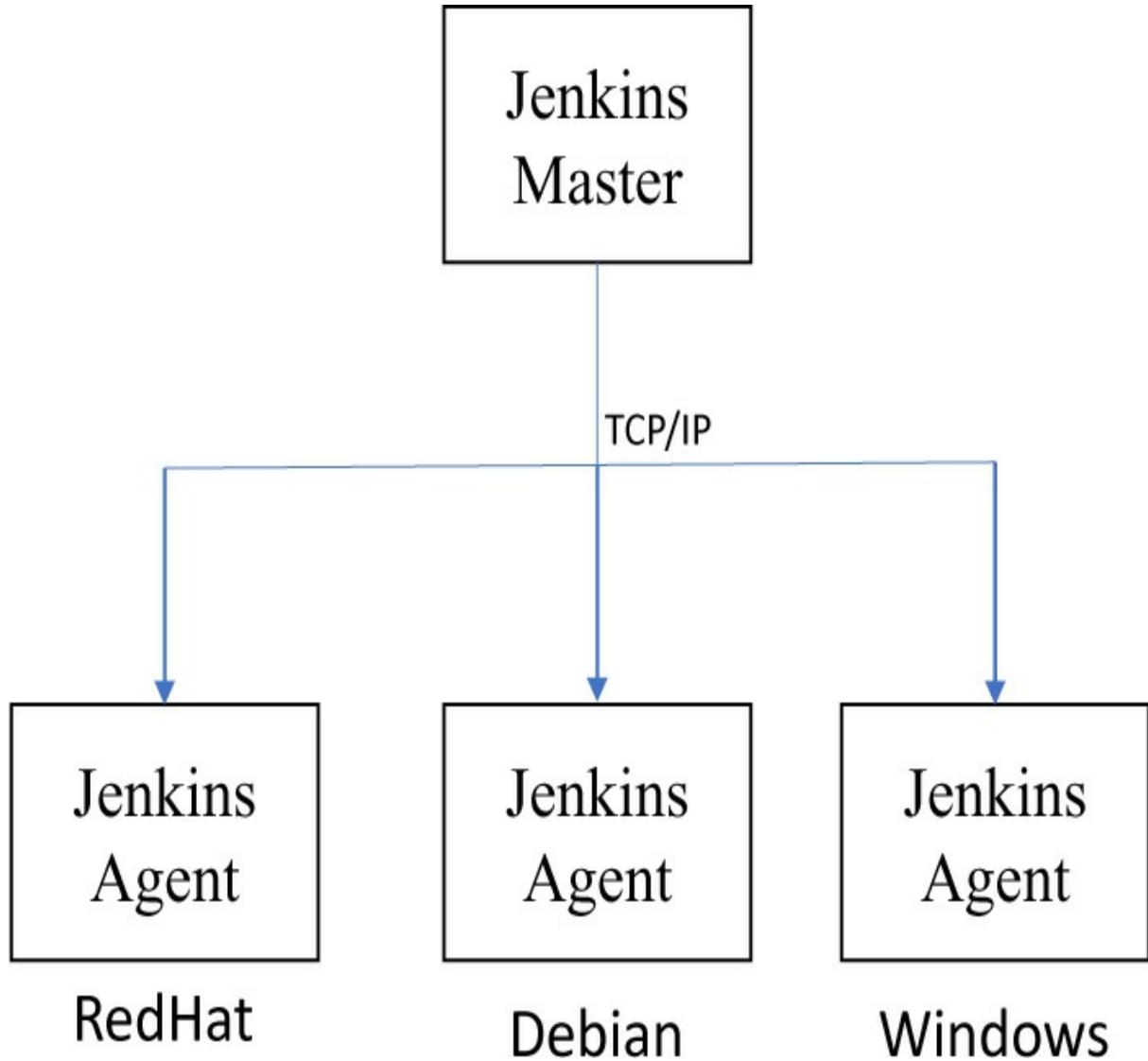


Figure 6-8 Jenkins Distributed Architecture

One of the main benefits of using Jenkins is the immense number of plug-ins it offers. These plug-ins enhance the native functionality of Jenkins and enable it to integrate with pretty much anything, including Git and Slack.

Jenkins Pipeline is a suite of plug-ins that implements a CI/CD pipeline. These pipelines in Jenkins are represented in a text file called `Jenkinsfile`. There are two formats for these pipelines:

- **Scripted:** These pipelines are older and use Groovy programming language syntax. Scripted pipelines are traditional pipelines that offer more control than declarative pipelines.

- **Declarative:** These pipelines use a simplified syntax (see [Example 6-17](#)). They are more structured and opinionated than scripted pipelines.

Example 6-17 Syntax of a Mock-up Jenkins Pipeline

```
pipeline {
    agent any

    stages {
        stage("Build") {
            steps {
                echo "Here we would build our code"
            }
        }

        stage("Test") {
            steps {
                echo "Here we would put our tests"
            }
        }
    }
}
```

In [Example 6-17](#), you will see is the keyword *agent*. This is a mandatory argument that tells Jenkins in which nodes to execute. (In this example, any of the available nodes can be used.) The *stages* argument is a mandatory argument that represents the logic separations where the work will be executed. Typically, there are three or four stages (such as configure, test, and deploy). Inside stages is *steps*, which is also a mandatory argument. It defines a sequence of instructions to be executed within a stage. [Figure 6-9](#) illustrates the pipeline shown in [Example 6-17](#). Keep in mind, however, that the start and end always exist and are not always represented in the Jenkinsfile.



Figure 6-9 Jenkins Pipeline Stages

Other keywords are available in a Jenkinsfile, but the one shown here are the minimum required ones.

In a NetDevOps environment, you can use the same concepts, and your steps will be network automation instructions (for example, Ansible playbooks). The pipelines you create can replace your manual workflows.

There are several ways to trigger defined Jenkins pipelines, including the following:

- Manually, using either the Jenkins GUI or API
- Automatically, based on changes in a Git repository

How to Build Your Own NetDevOps Environment

This section describes how you can build a small NetDevOps environment using Jenkins and Git as well as Ansible. This NetDevOps environment will allow you to trigger a pipeline that tests and configures network equipment (in this case, routers).

NetDevOps Environment Requirements

First of all, you should have a Git repository for storing your code and configurations. If you are using a public repository (for example, GitHub), keep in mind the importance of not publishing anything confidential, such as your credentials or important configurations.

We cover Ansible in depth in [Chapter 4, “Ansible Basics,”](#) and [Chapter 5, “Ansible for Network Automation,”](#) so by now you are familiar with it. You will use Ansible to interact with the devices.

You can install Jenkins in a virtual machine or container, as previously mentioned. If you are using a single Jenkins node, make sure to install Ansible on the same machine as it will be the one performing the work.

Besides these tools, you also need network devices to connect to, as defined in an Ansible inventory file.

When you have these components in place, you are ready to start. You should upload to your Git repository all your components.

Your directory structure in Git should look something like the one in [Example 6-18](#) (although is not mandatory to have this structure). In this case, the device's hostname is switch_01, and the configuration file has the same name. Furthermore, you have two Ansible playbooks—one to push your configurations (configure.yml) to the network devices and another to test the results (testing.yml)—as well as an inventory file named hosts. You could do verification and testing before making configuration changes to the network devices; however, for the sake of simplicity, this example covers only postchecks.

Example 6-18 Sample Directory Structure of Git Repository for NetDevOps

```
$ tree
.
├── configurations
│   └── switch_01.yml
└── playbooks
    ├── configure.yml
    └── testing.yml
```

[Figure 6-10](#) shows a reference architecture for this example, using a single node.

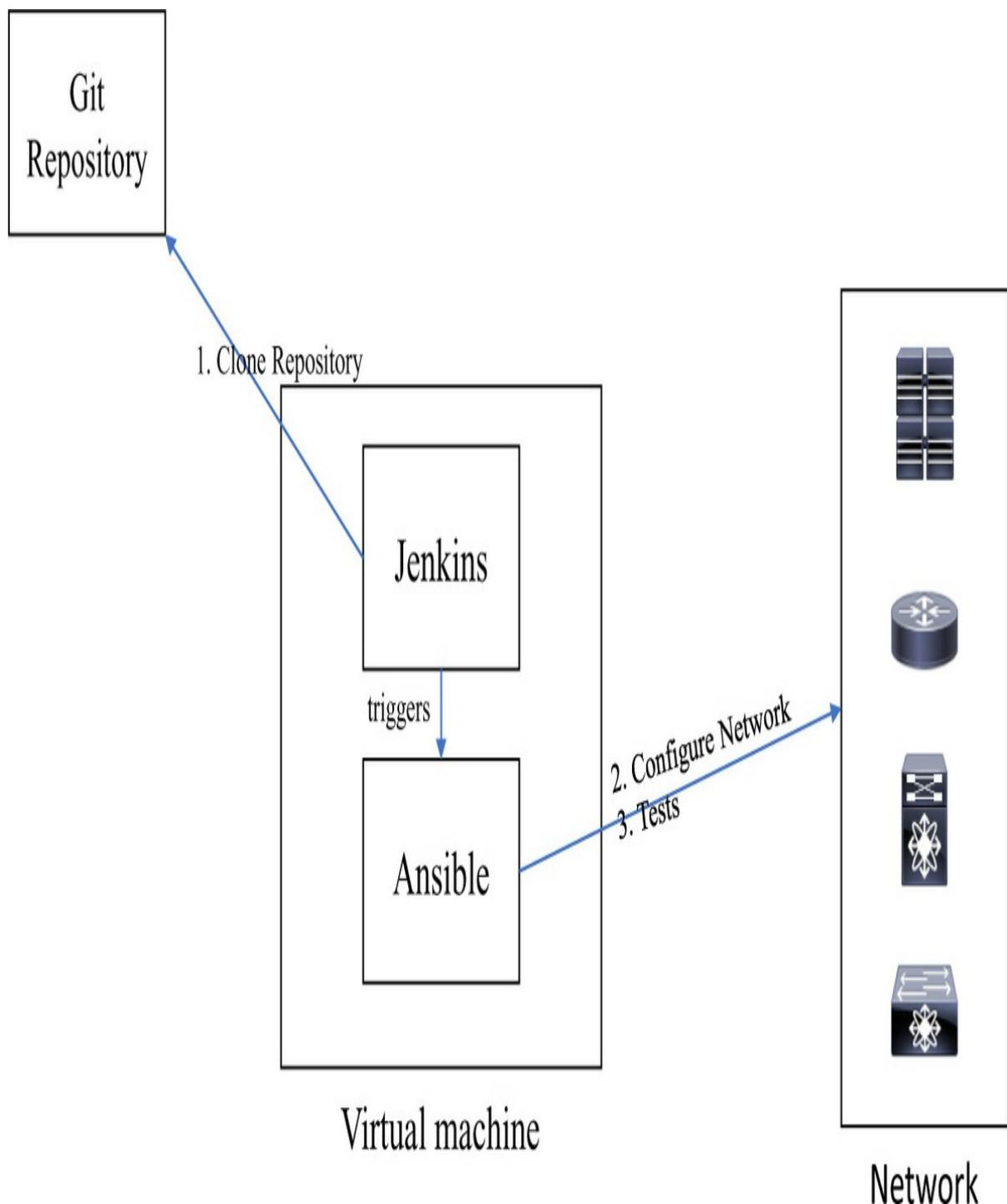


Figure 6-10 Jenkins Test Architecture

NetDevOps Stages

The pipeline will comprise only three stages:

Stage 1. Clone the Git repository that holds your Ansible playbooks, inventory, and device configurations.

Stage 2. Execute the Ansible playbook that make the configuration changes.

Stage 3. Run a playbook to verify that it is configured and working as expected.

A scripted pipeline could look like the one in [Example 6-19](#). You need to replace the Git and Ansible variables in this example with your own.

Example 6-19 Three-Stage Jenkins Scripted Pipeline

```
node {  
    stage('Git Checkout') {  
        git branch: 'main',  
            credentialsId: '#OMMITED#',  
            url: 'https://wwwin-github.cisco.com/Ivpinto/netdevops'  
    }  
  
    stage ('Configure routers') {  
        sh 'ansible-playbook -i hosts playbooks/configure.yml'  
    }  
  
    stage ('Test network') {  
        sh 'ansible-playbook -i hosts playbooks/testing.yml'  
    }  
}
```

Note

You may have several steps within a stage instead of a single one.

[Example 6-19](#) shows that this example uses a very simple playbook to push a template configuration file using the *ios_config* Ansible module.

Note

If you need a refresher on how to build an Ansible playbook to configure or test devices, refer to [Chapter 5](#).

Example 6-19 Ansible Configuration Playbook Using Configuration Templates

```
$ cat configure.yml
---
- name: Ansible Playbook to configure a device
  hosts: ios

  tasks:
    - name: Configure device from template file
      cisco.ios.ios_config:
        src: "configurations/{{ ansible_net_hostname }}"
```

Notice in [Example 6-19](#) how you indicate which configuration file to use by linking it to the hostname.

With this pipeline configuration at the ready, you can go to the Jenkins UI, click **New Item**, click **Pipeline**, and give the pipeline a name. Finally, you paste the configuration into the pipeline script definition and click OK.

You would now have a working NetDevOps pipeline! A successful execution looks as shown in [Figure 6-11](#).

Note

By default, Jenkins keeps all the logs from the execution. You can use these logs to debug any failure.

Stage View

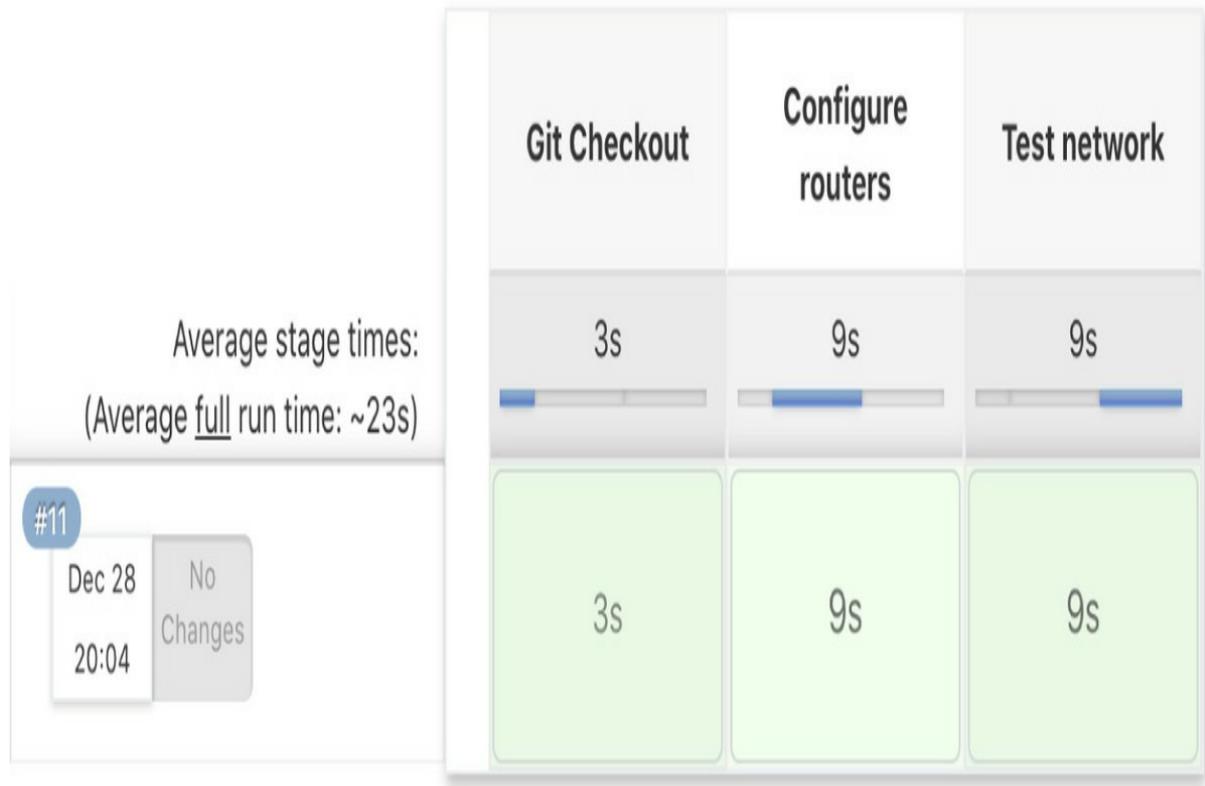


Figure 6-11 Jenkins Pipeline Stages

Based on the way the pipeline was created, you must trigger this pipeline manually after committing a configuration change to Git. However, Jenkins has a plug-in that allows builds to be triggered automatically. Git commits are commonly used triggers. When you modify a configuration file in your Git repository, it triggers the pipeline and pushes that configuration to the network devices. This is a more advanced use case.

In a more complete pipeline, you would first deploy changes to a test/lab/preproduction network, and only if they worked there, you would apply those changes to the production network.

Tip

To test what you have learned so far, think about how you would enhance the functionality of this simple pipeline and make it

interact with two isolated networks. Keep in mind that you can use the same playbook but two different Ansible inventories.

NetDevOps Operations

You can achieve a multitude of different operations by using CI/CD and automation. Testing is one of them.

Testing is an extensive topic, and it is incredibly important when it comes to NetDevOps. You should have tests beyond simply **ping** tests and tests to verify that configuration commands were entered correctly in a device. You need tests that reflect real users' operations in order to make sure the network will be operational for them.

You can use physical devices or virtual devices for testing. There are upsides and downsides to both approaches:

- Virtual devices are elastic. It is quick and easy to spin them up or down.
- Virtual devices are cheaper.
- Virtual devices typically do not function well when testing data traffic.
- Virtual devices are not an exact match to your network production environment.

There are many platforms that virtualize network devices, including Cisco Cloud Modeling Labs, GNS3, Quagga, and EVE-NG. Your choice depends on your requirements. You might also choose to do your testing using physical devices.

CI/CD pipelines can be very extensive. The following are a few operations you might want to add when building your own pipelines:

- Linting to verify whether your code has errors (Linting can be applied to most languages, including YAML.)
- Building configurations from templates instead of statically
- Using containers to manage dependencies
- Rolling back steps for failed stages

Finally, CI/CD pipelines can help you with tasks other than configuring and

testing network changes. The following are some examples:

- Network health status reports
- Network compliance reports
- Network testing

Although these tasks are not central to NetDevOps, they enable you to combine automation and CI/CD pipelines for networking use cases.

Case Studies

The three case studies in this section are not all standard and typical NetDevOps deployments, where the infrastructure is represented in a text format in a source control repository and changes to it are tested and eventually pushed to production by a CI/CD server using automation. Some of these cases have a slightly different twist but are on the path to NetDevOps. Currently, most customers are on the path and not yet ready for the full NetDevOps scenario; even so, they can get value from these practices. Hopefully, you will be inspired by these case studies.

Pipelines in One of the World's Largest Banks

A bank with worldwide presence had many issues related to agility. It was simply too slow at pushing out changes, and it was affecting their time to market. Banks are notoriously slow when it comes to innovation, often due to regulatory compliance or simply mindset. The bank in this case wanted to address this.

The bank started out by identifying exactly what it wanted to achieve more quickly: testing. The bank wanted to be able to test a new application in a production-like environment without having to go through a huge process of requests and approvals. Also, it wanted the test scenario to be very close or equal to production because it had faced many situations in the past where the internal testing had passed successfully but the rollout to production had unveiled new issues.

To address this initial requirement, the bank developed a separate network

named Play that was a small-scale replica of the production environment with fewer controls. This new environment was accompanied by CI/CD pipelines.

Using the pipelines was the only way of interacting with Play. The bank could not use SSH to access a switch in this environment. By removing all access to the network and using pipelines as the only interface, the bank changed the way workers saw a network. In essence, configuration drift could no longer exist.

There were plenty of pipelines:

- **End-to-end pipelines:** These pipelines were aimed at developers, providing menus for choosing the location of code (the source control repository), the operating system to use, the type of security needed for an application, and so on. It was much like how deploying a virtual machine in the cloud works. The pipelines implemented the necessary logic to achieve the goals and abstracted that logic from the user.
- **Task pipelines:** These pipelines, which were implemented using Ansible and Python, were aimed at the networking folks. The task pipelines achieved tasks such as creating VLANs, advertising networks in routers, changing the SNMP community, and adding more storage on a VM.
- **Compliance pipelines:** These pipelines were for generating reports and metrics from the network. These pipelines mostly used Python to retrieve the necessary information and output reports.

The new process was a success. Developers were happy that they could test their applications in a production-like environment, and network engineers were pleased that they did not have to know the specific syntax for commands or even which IP addresses the devices had. Most importantly, every action on the network was tested. All pipelines had a test stage and a rollback prepared in case the tests did not succeed.

Later, the bank decided to go for the next step. It migrated its production network to the same ideology. It removed direct access, automated all tasks, and allowed users to interface only through pipelines. This was a 5-year effort.

The bank now has hundreds of pipelines. A new hire learns how to use these pipelines instead of how to connect to complex network and computing

topologies.

This is a great example of a spin on NetDevOps. The bank uses IaC to represent the network state, Git to store all generated code, Ansible to configure devices, Terraform to provision instances in the cloud, plenty of Python scripts to achieve specific functions, and Jenkins as a CI/CD server to tie it all together.

The bank has not had an outage in over 4 years.

New Technology Trainings at a Fortune 50 Company

A huge IT company held a lot of trainings, both internally and externally. Each of these trainings required a lot of hardware so trainees could follow lab guides.

The company had dedicated hardware for the trainings, but it was spending almost as much time resetting and reconfiguring the training pods as actually giving trainings. This was unacceptable from a business perspective, so it had to be addressed.

The company determined that this is what it took to prepare a training pod:

Step 1. Reset the pod to its initial state.

Step 2. Upgrade the hardware and software to the desired versions.

Step 3. Configure required features.

Step 4. Verify that the pod is working as expected.

Depending on the training pod, this workflow could be applied on a single device or on several. Typically, a pod had more than five devices (of different types, such as routers, switches, wireless controllers, and firewalls).

As you might image, handling these steps manually was a burden. The company started by automating some of these tasks:

- Using Ansible to configure the devices.
- Using Ansible to upgrade/downgrade versions.

- Testing functionality using automated test suites.

Automation substantially reduced the burden. However, a person still had to trigger all of these tasks sequentially and wait for them to finish in order to trigger the next one. This is where NetDevOps came in.

The company developed CI/CD pipelines for each training so that, at the click of a button, it could have the hardware configured for one architecture/technology or another. These pipelines had one for each major action (for example, resetting to the initial state), and within a stage, they could call automation tools such as Ansible to perform the needed tasks.

Now all the company needs to do is trigger the pipelines before a training, such as the night before, and the next morning, everything is ready. If something goes wrong, the company can look at the full report from the testing, which includes actionable insights.

The company took this solution a step further to solve another problem. After attending a training, people were saving the credentials and then using the pods later. The company wanted to solve this problem. First, it developed a pipeline for reservations that generates a random-access key and rotates it when the cleaning pipeline is triggered. Here is a summary of this solution:

- One pipeline per training type per pod
- Four stages per pipeline (reset, upgrade, configure, test)
- Automation in each stage
- Time-based triggers for the pipelines (based on reservation times)

There is no testing step before configuring the actual devices because there is no criticality involved. The company is using CI/CD to orchestrate the automation that had previously been built.

New Service Implementations in a Tier 1 Service Provider

Service providers are used to implement new services, such as installing a router in a customer's house. The specific service provider in this case study has a global customer base. It was having trouble being agile, especially

when it came to implementing new VPN services. As you know, a VPN configuration touches several components, among them the PE and CPE routers. These can be geographically separated by thousands of miles.

Most service providers have some type of orchestration workflow with templates in order to push the configurations to the devices. When the time comes to activate a new service, an automated tool or a person pushes the configuration templates with specific values to the required devices. This service provider had such a workflow in place. However, it was having trouble with verification.

In 10% of the new service implementations, after the configuration of a new service, the service provider had issues. This was a very high percentage, leading to many customer complaints and hundreds of hours of troubleshooting per year.

The service provider decided to implement pipelines for its network—specifically for new service implementations.

The service provider created one pipeline per service type. Each pipeline consisted of the following steps:

Step 1. Preparation. The service provider gathers the template information required, along with user-provided parameters, and verifies against a rule-based system (linting).

Step 2. Configuration in a virtual topology. The service provider creates a virtual network and pushes configurations to those virtual devices.

Step 3. Verification in a virtual topology. The service provider verifies that the configuration is correctly pushed and makes some tests possible in virtual devices (for example, **ping** tests).

Step 4. Pre-verification in production. The service provider gathers baseline information about how the process was working before the change, so it could compare that information to a known good state by the end.

Step 5. Configuration in production. The service provider pushes the configuration to the devices in the production network.

Step 6. Verification in production. The service provider compares the state to

the previously collected baseline and verifies end-to-end functionality. This testing stage is more in-depth as some of the testing required is not supported in virtual infrastructure.

There is a rollback step defined in case the tests fail. This step reverts to the old configuration but saves logs, traces, and configurations from the failure so it can be investigated later without causing network impact. If a failure occurs in the virtual topology (step 3), the configurations are not pushed to the production network. These pipelines are triggered automatically from a source control repository. This repository was another addition to the service provider's adoption of NetDevOps. The service provider migrated its configurations to the source control repository. The service provider keeps its configurations there, and it has also abstracted services and represent them in a text IaC format.

[Example 6-21](#) highlights a possible addition of a new Layer 3 VPN service for this service provider. When a new service must be configured, an operator can modify a file to represent the services state, and that modification can trigger the pipeline to eventually reflect the change in the production network.

Example 6-21 Simplified Network Service Abstraction in Text

```
{ "L3VPN":  
  [  
    {  
      COLOR: "red",  
      FROM: "Portugal",  
      TO: "Spain",  
      BW: "10G",  
      Redundancy: "None"  
    },  
    {  
      COLOR: "blue",  
      FROM: "Portugal",  
      TO: "Germany",  
      BW: "1G",  
      Redundancy: "None"  
    }  
  ]  
}
```

```
        ],
    "L2VPN": [
        {
            FROM: "Portugal",
            TO: "Spain",
            BW: "10G",
            Redundancy: "None",
            vlans: "22,44"
        }
    ]
}
```

This type of abstraction allows the operator to have a simplified view and does not require in-depth knowledge of networking to enable a new service. It is truly a network as code implementation.

Summary

By now you should see that there is no magic behind NetDevOps—just practices that worked in the software industry, became best practices, and made their way to networking.

This chapter cover what NetDevOps is, how it evolved from DevOps, and why you should care. This chapter also describes some of the most famous CI/CD tools, and you have seen that CI/CD servers provide a nice way of replacing manual workflows with automated reusable ones.

This chapter describes source control tools where you should store your text configurations and code so you can benefit from history, comparisons, and other benefits.

This chapter also shows you how to create your own network as code pipeline.

This chapter finishes with three case studies showing companies that have used NetDevOps techniques to create business value and enhance their operations.

Review Questions

You can find answers to these questions in [Appendix A, “Answers to Review Questions.”](#)

- 1.** True or false: NetDevOps is a software development methodology.
 - a.** True
 - b.** False
- 2.** Which of the following is *not* a benefit of infrastructure as code?
 - a.** Configuration drift
 - b.** Ease of replication
 - c.** Repeatable deployments
 - d.** Ease of distribution
- 3.** Which of the following is *not* a phase of DevOps?
 - a.** Release
 - b.** Plan
 - c.** Create
 - d.** Test
- 4.** Which of the following are Jenkins pipeline definitions? (Choose two.)
 - a.** Declarative pipelines
 - b.** Generic pipelines
 - c.** Scripted pipelines
 - d.** Basic pipelines
- 5.** Which of the following is an example of a CI/CD server?
 - a.** Jenkins

b. Git

c. Ansible

d. Terraform

6. What type of source control system is Git?

a. Distributed

b. Local

c. Central

d. Principal

7. True or false: One way of scaling a Jenkins deployment is by adding more agents.

a. True

b. False

8. In Git, which of the following do you use to create an independent line of development from an existing repository?

a. git branch

b. git commit

c. git merge

d. git add

9. In Git, what command must you use to roll back to any previous commit?

a. git checkout

b. git commit

c. git log

d. git add

10. When designing a Jenkins pipeline, what is the maximum number of steps you can define in a single stage?

- a.** 1
- b.** 10
- c.** 100
- d.** Unlimited

Chapter 7. Automation Strategies

So far in this book, you have learned a lot about automation, from types of automation, to use cases, to tools to help you fulfill the use cases. You have learned about data and what you can accomplish by using it. You have learned about using Ansible as an automation tool and implementing playbooks. You have even learned about using NetDevOps pipelines to automate the delivery of end-to-end activities in a network.

This chapter covers strategy, which is notoriously missing in many companies today but is key for success. This chapter looks at what an automation strategy is, what it consists of, and how it relates to other types of company strategies. As you read this chapter, you will begin to understand the importance of having an automation strategy to guide you on your journey.

This chapter ends by summarizing an approach you can use to build your own automation strategy with a focus on network automation.

What an Automation Strategy Is

Your company most likely has an organizational strategy, a financial strategy, and a go-to-market strategy, but not an automation strategy. Why is that? Well, although automation itself is not new, many companies don't know a lot about it. Many companies have been forced by the market into adopting automation without planning it. There are many ways to define what a strategy is, but the basic idea is that a strategy reflects where you are, where you want to be, and how to get there.

An automation strategy is supported by five pillars, as illustrated in [Figure 7-1](#):

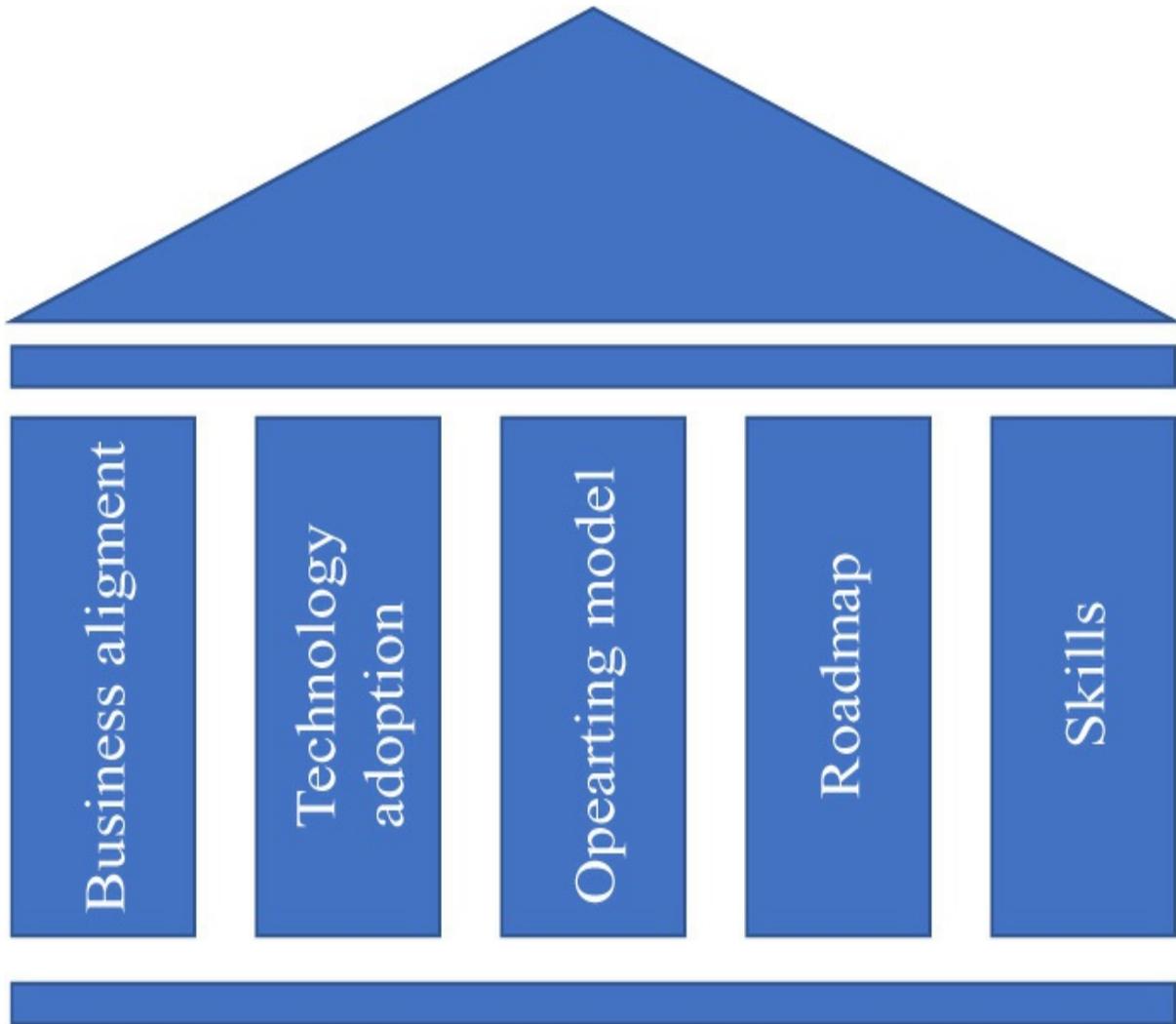


Figure 7-1 Automation Strategy Pillars

- **Business alignment:** An automation strategy should be in line with an organization's overall business strategy. The business strategy reflects the mission of the company, and therefore all other strategies should align to it. Automation should always work as an enabler to the overall business.
- **Technology adoption:** An automation strategy should reflect what technologies are in use and what technologies the company will be adopting in the future. Having this information clearly stated helps avoid sprawl in tools and technologies. Furthermore, it helps facilitate automation—for example, by enabling you to choose technologies with good APIs.

- **Operating model:** An automation strategy should reflect the operating model (owners and maintainers) for the automation tools and projects. Setting clear responsibilities is paramount to keep things under control and maintained; this also helps controlling project sprawl.
- **Roadmap:** An automation strategy should document what is ahead, including tools that will be adopted, upcoming new projects, and/or decommissions. It is not an official roadmap document, but roadmap perspectives are reflected in a strategy.
- **Skills:** From the automation strategy document, it should be clear what skills are needed for the future. Does the organization already have those skills, or do they need to be built/hired? Skills are tightly correlated to culture, and automation commonly requires a cultural shift.

You can produce a strategy document that reflects these five pillars by defining four concepts:

- **Goals:** High-level purposes
- **Objectives:** Specific outcomes that define the goal that are measurable (that is, linked to KPIs)
- **Strategies:** How to achieve the goals from a high level
- **Tactics:** Specific low-level strategy actions to follow

These concepts can be illustrated as an inverted triangle, as shown in [Figure 7-2](#), that goes from high-level goals all the way down to tactics.



Figure 7-2 Goals, Objectives, Strategies, and Tactics

The goals are broad statements that typically reflect desired destinations. They are often the “what” that you are trying to achieve. Although a goal reflects an end state, it is not actionable; that is the role of objectives. Some examples of goals are being the quickest at deploying in the market segment, having zero-emissions data centers, and having a closed-loop automated business.

Objectives are actionable, specific, and measurable. Furthermore, they should be realistic and achievable. They define what needs to be done to achieve the previously defined goals; therefore, objectives do not exist in isolation but are

always linked to goals. The following are some examples of objectives linked to the goal of having zero-emissions data centers: decrease dependence on fossil fuels by 10%, increase use of green energy sources by 20%, reduce data centers' resource sprawl by 15%.

Strategies are high-level plans to follow to achieve goals. They specify, from a high level, how you are going to fulfill your intentions. Examples of strategies linked to the objective of reducing a data center's resource sprawl could be monitor server utilization and use a cloud elastic model.

Finally, tactics are specific low-level methods of implementing strategies. Examples of tactics linked to the strategy of monitoring server utilization could be deploying a new monitoring system using Grafana and creating new metrics of device utilization such as CPU utilization.

Assessment

Before you can create a strategy document, you need to assess where you are now. An assessment focuses on identifying the current state of affairs, and it can be a lengthy process. During the assessment, you should aim to identify which technologies you are already using and to what extent. It is good to assign a maturity level on a fixed scale (for example, from 0 to 10) so you can effectively compare the different tools you might be using.

Note

During an assessment, you are likely to discover all kinds of technologies and tools that you were not aware existed in the company. This is normal, and you shouldn't let it discourage you. It is also common to discover a great variation in tool use across different departments in the same company.

An assessment should also consider the current industry landscape—not just your company's. This is especially important for the next phase, which involves determining where you want to be. (Later in this chapter, you will see an example of the process of building a strategy document.)

When the assessment is finished, you should know where you are. It is then

time to define where you want to be. When defining where you want to be, there are many aspects to take into consideration, including these:

- **The costs and benefits:** You might wish for an end-to-end automated network environment, but that might be financially out of reach if you currently have no automation.
- **Skills available:** Do you have the skills in-house to achieve your goals? Will you need to train your people? Will you need to hire other people?
- **Prioritization:** You might come up with tens of destinations, but out of all those, which are the best for you? A prioritization exercise is critical when defining a future state. You might want to save the destinations you discard for strategy updates in the future.
- **Possible risks:** If there are identified risks involved, you should take them into account and document them. Risks can play a major role in prioritization.

With these aspects in mind, you define goals, then objectives, then strategies in terms of automation and document what you come up with in the strategy document. Automation does not mean only network automation; the automation strategy document should have a broader scope. (However, because this is a book on network automation, we focus solely on network automation in this chapter.)

Defining tactics—which tell you how to get where you’re going—is often the most technical part of this exercise. You do not need to define when tactics will be achieved or the detailed steps to achieve them. You will do those things as part of an action plan, which is defined at a later date, as a byproduct of the strategy. (You will learn about action plans later in this chapter.)

KPIs

The four concepts just described—goals, objectives, strategies, and tactics—are the minimum components of an automation strategy. However, other concepts are commonly included as well. An important concept to add to an automation strategy is key performance indicators (KPIs). Sometimes companies list KPIs in a separate document and mention that document in the

automation strategy. However, including them directly in the automation strategy is helpful because it prevents people from getting lost between different documents.

A KPI is a quantifiable measurable value that can demonstrate how well or how badly you are doing at some activity. KPIs are key to tracking your objectives; therefore, even if you do not add them to your automation strategy document, your objectives should somehow link to the company's global KPIs.

There are high- and low-level KPIs. High-level KPIs typically focus on the overall business performance and result from the work of multiple departments and individuals. Low-level KPIs focus on more specific processes and are influenced by specific individuals.

An example of a high-level KPI could be a measure of the revenue growth rate of a company, which is influenced by the performance of all the departments. A low-level KPI could be a measure of services deployed by a team under the operations department, solely influenced by those team members (which means it is more traceable and actionable). Creating a strategy for network automation typically means defining low-level KPIs.

The most common mistake with KPIs is blind adoption. Companies and departments often copy KPIs from other companies or departments without thinking about whether they accurately reflect their business or operations. This often results in failure to positively impact results and leads to abandoning the KPIs because the resources put into measuring are effectively wasted. It is paramount to have KPIs that resonate with your company's objectives; this is the only way they can be effective.

Another common mistake is the form of communication of the KPIs. KPIs should be easily accessible, such as on a dashboard, and communicated often so individuals can understand where the company is and where it wants to go. Often KPIs are hidden in long documents reported at the end of fiscal year or fiscal quarter, and people are not aware of the KPIs and therefore do not actively work toward them. Finding a good communication mechanism is key.

A third mistake is adopting KPIs that are hard to measure, not actionable, too vague, or subjective, which can lead to discussions and conflicts. The

following are some examples of KPIs with these characteristics:

- Make the scripts tidier
- Use fewer automation tools
- Improve coding practices

A fourth mistake, which is a very dangerous one, is adopting KPIs that are easy to manipulate. With such KPIs, people who want to be recognized may abuse the KPIs to elevate their performance without actually having a positive impact on the business. For example, if a KPI measures the number of software releases, a team could start making very small releases (for example, for simple bug fixes) just to influence the KPI instead of following the normal life cycle of releasing a version only after some number of fixes. In a case like this, although the KPI might make it look like the business has improved, it actually hasn't. The problem is even worse if you link incentives to KPIs because the incentives might motivate individuals to cheat. KPIs exist to measure where you are in relation to where you want to be; they are navigation tools. Linking incentives to them transforms KPIs into targets.

Yet another mistake is evaluating KPIs with tainted data. For example, when you are measuring software compliance and you only take into consideration the software version of the network equipment in the HQ and neglect branches because they are managed by some other team, you are using tainted data. You should make sure you have high-quality data.

Another common mistake is measuring everything or measuring everything that is easy to measure. KPIs require data, and it is easiest to measure something you already have data for. KPIs should measure only what is necessary. Gathering and storing data can be expensive, and KPIs should be relevant.

Now you understand the common pitfalls, but how do you create good KPIs? We focus here on network automation, but the same principles discussed here apply to other types of KPIs as well.

The secret sauce is in the name: *key*. KPIs should exist only in relationship to specific business outcomes and critical or core business objectives.

There are many approaches to defining a KPI, but I find that answering the following seven questions is often the best way:

1. What is your desired outcome? (This is the objective you are starting with.)
2. Why does this outcome matter for the company/department?
3. How can you measure this outcome in an accurate and meaningful way?
4. How can people influence this outcome?
5. Can you tell if you have achieved the desired outcome? If so, how?
6. How often will you review the progress made?
7. Who is ultimately responsible for this outcome?

Let's say you are a service provider, and your objective is to decrease your average time to deploy new network functions, such as a new firewall type or a new SD-WAN router. The following are possible answers to the seven questions:

1. Decrease by 10% the time it takes to deploy a new version of a network function.
2. Decreasing the deployment time will allow us to acquire customers with stricter SLAs and therefore increase business profitability.
3. We can use scripts to measure the time it takes for a new deployment to happen.
4. Adoption of network automation techniques can impact the time required as currently we are doing it manually.
5. We can compare the previous time of deployment to the current to determine whether we have achieved the objective.
6. Progress shall be reviewed on a quarterly basis, so we can make adjustments in regard to found challenges/lessons learned.
7. The network operations team should be responsible for this KPI.

By providing answers to the seven questions, you can define a KPI. In this case, the KPI is related to the time it takes to deploy a new network function. You should evaluate whether a newly defined KPI makes business sense, and you can do so by answering five questions, commonly known as the SMART (specific, measurable, attainable, relevant, time-bound) criteria:

1. Is your objective specific?
2. Is your objective measurable?
3. Is your objective attainable?
4. Is your objective relevant?
5. Does your objective have a time range?

In our previous example, your objective is specific: decrease by 10% the time for new deployments. It is also clearly measurable, as it is possible to measure the time, in minutes, it takes to deploy a new network function.

Attainability can be difficult to understand as you need to factor in history and context. If you were doing everything in this example manually—for example, pushing router configurations—a 10% increase would be attainable by switching to automated scripts.

The objective in this example—acquiring new customers—is relevant. You would need to factor in the cost of automation and the benefit that the new customers bring.

Finally, you have not set a time range. You simply set a quarterly review of the KPI, and it would be good to limit your objectives in time to something that is achievable and realistic, such as 6 months for the 10% decrease.

In summary, you should always align KPIs to business objectives. This correlation should always exist. You shouldn't have objectives without KPIs or KPIs without objectives. You may have several KPIs for a single objective, however.

Other Strategy Documents

Now that you understand what are KPIs, why they matter so much, and where you should have them in your automation strategy, let's circle back to the automation strategy document and look at the other sections it may have. It can and should have other strategies attached that link to automation. Two strategies commonly linked to the automation strategy are the data strategy and the financial strategy. These should exist as sections of the automation strategy, highlighting relevant parts of those strategies for automation.

A data strategy is tightly coupled with an automation strategy, as you saw in Chapter 2, “[Data for Network Automation](#),” and Chapter 3, “[Using Data from Your Network](#).” Automation uses data to derive actions and insights and to understand the health of the network. It is common to already have a data strategy in place, and relevant parts of it should be highlighted in a section of the automation strategy document. It is also possible that the data strategy needs to be modified to accommodate some of the automation strategy goals and objectives.

Why would you want a data strategy? What problems does it address?

Data has become the most important asset of many companies. A data strategy attempts to get the most out of the data you have available or that you can collect. Here are some of its goals:

- **Steer you in right direction:** There are many data sources. In the past, companies did not pay much attention to them, and today they are flooded by data. Too little data is not good, but too much is equally bad. A data strategy helps you understand what data you need to achieve your goals and where will you collect that data. It also helps ensure the quality of your data.
- **Help you discover how to use the data:** After you’ve collected all the data you need, you need to know how to make use of it. What stakeholders need the data, in what format, and how can you create value from the data?
- **Monetize data:** Data can be very valuable. Some companies are not aware of the valuable asset they have in store. To build a data strategy, you need to make an inventory of the data you have available, which helps you identify any potential monetization. Monetization in this context does not mean selling the data; it can simply mean using the data to do more effective marketing or using the data to power more intelligent customer service.
- **Meet compliance requirements:** You might be subject to multiple data regulatory compliance laws and standards, depending on your region and industry (for example, PCI-DSS, GDPR, HIPAA). A data strategy will help you identify what laws and standards you must comply with and avoid serious consequences.

Now that you are convinced that a data strategy is important, what *is* a data strategy? The idea behind a data strategy is to make sure all data resources are accounted for. Data is no longer a byproduct; it's a critical asset that enables decision making. Just like an automation strategy, a data strategy defines a common set of goals and objectives. However, it focuses on data and ensuring its effective and efficient use. Furthermore, a data strategy defines methods and processes to manage, store, change, and share data in a company. It should highlight the data collection methods and policies for network data along with where it will be stored and how it will be governed and used.

From a typical data strategy, you will be able to extract the following:

- **Data needs:** What do you need data for? What are the goals? The data needs should be the first thing to be identified and highlighted, as they are the starting point for a data strategy.
- **Data sources and gathering methods:** You need to understand where that data you need lives and how you can retrieve it. There are many different ways of gathering data (refer to [Chapter 2](#)). When defining a data strategy, you need to compare the various methods for each data resource required, and you need to document the results. By consulting the data strategy, a reader should be able to know what method he needs to use for a specific data resource. It is also important to highlight where and how the data will be stored.
- **Data insights generation:** After you have collected data, you need to turn it into something useful to get insights. As part of the data strategy, you need to enumerate methods to generate insights. For example, if you are collecting measurements of the temperature of your network devices by using SNMP polling, you might transform this data measurement into an alarm (insight) by using management software that has a threshold configured.
- **Data infrastructure:** When you reach this section, you already know what data you need, how you will gather it and from where, along with how you are going to transform it to meet your needs. This section should illustrate what technology stack and infrastructure are needed to support the whole process; it could be Git, Ansible, Python, or anything you choose.

- **Data governance:** All companies that store data may have to comply with certain regulations. For example, in the payment industry, companies must comply with PCI-DSS, and in the health care industry, companies must comply with HIPAA. In general, the more confidential the data, the more regulations it is subject to. Companies need governance processes for their data. Some common governance aspects are who has access to what data, how long to keep certain data, how the data be protected, and where to store the data.

A data strategy needs to include KPIs to effectively measure progress and results, just as in the other strategies. Finally, in case it was not already obvious, the data strategy needs to be aligned with the overall business strategy.

It is critical to consider the data strategy when building an automation strategy because it can become a blocker. For example, say that a company is not allowed to store any data, including network data, for over 30 days, but for its automation strategy, the company had envisioned creating a machine learning model that would require a year's worth of historical data. Clearly, the company would not be able to achieve this goal because of the company's data strategy.

Similar principles apply to the financial strategy. Automation costs money, in terms of both labor and products. Make sure you are aligned to the overall financial strategy and highlight any relevant information on a section of the automation strategy document. For example, the financial strategy could indicate that the company is cutting expenditures on data storage. If this were the case, you wouldn't want to define goals that require storage of even more information than you already store.

Note

Although many companies do not have data or automation strategies in place, almost all companies already have a financial strategy in place.

Summary

To summarize, an automation strategy is a document that includes goals, objectives, strategies, and tactics, along with sections linking to other relevant strategy documents, such as the data strategy. It is a living document that can and should be iterated upon when changes are required. Iterations can happen for many reasons, including the following:

- Industry events
- Other company strategy changes
- Failure to meet KPIs

Furthermore, an automation strategy is a document framed in a time range. It should be objective, concise, and easy to understand by all stakeholders.

By having an automation strategy document, you and your company have a path to follow. Having clear destination will prevent you from wandering around aimlessly.

It is very important that an automation strategy document be well understood by all stakeholders, from higher management to operators and everyone in between. If everyone shares a common view, they can work toward the same destination.

Note

Try to keep your automation strategy document short and relevant. Most stakeholders will not read and cannot understand 50-page documents.

Why You Need an Automation Strategy

We have seen many companies adopt automation tools and processes only to decommission them in a month's time. Other companies have tens of similar yet different tools, multiplying the efforts and investments of their workforce. Why? Because they were adopted without a vision or goal but as short-term

solutions to various problems.

Having a strategy is similar to having a plan. With a strategy, everyone can read and understand what are you trying to achieve and how. A strategy allows you to do the following:

- **Remove ambiguity:** There is no ambiguity if you have a well-defined strategy. When they know that the actual goal is to provide the best user experience, Team A will no longer think that the goal is to make the service run faster by reducing computational complexity, and Team B will no longer think that the goal is to reduce latency.
- **Communicate:** A strategy communicates to everyone the intentions of the company in that area—and this is why it is important that the strategy be a document that is well understood by all stakeholders. It is not a document only for higher management; it is a document for everyone.
- **Simplify:** Your company can simplify business operations when everyone knows what you want to achieve, using which tools, and during what time. You prevent teams from creating their own solutions to what they think is the goal. However, creative thinking is still important. The strategy functions as a guardrail to keep projects on track.
- **Measure:** A strategy allows you to understand if something is working. If the metric was to reduce your deployment times by 10%, did you achieve it? Measurement makes KPIs valuable.

In summary, you need an automation strategy because you want to achieve something, and achieving a goal is easier with a plan.

How to Build Your Own Automation Strategy

Now that you are convinced you need an automation strategy, how do you start? Well, we need to start by assessing where you are so that you can build your goals, objectives, strategies, and tactics to define where you want to be.

First, gather the key stakeholders in your company. A strategy is not defined by one individual but rather by a diverse group, which may consist of technical and nontechnical folks. With this type of group, decisions and assumptions can be challenged, resulting in better and common goals. In a smaller organization, the group might consists of only two people. Nonetheless, two is better than one.

This section is focused on the network automation part of your automation strategy. Your document should also encompass other automation efforts, such as business process automation.

Assessment

An assessment of the current state—where you are—is the first activity to take place. In this section, we elaborate on how you can do this yourself. Typically, an assessment involves the following stages:

Stage 1. Create/update an inventory of automation tools/scripts.

Stage 2. Create/update a list that identifies needs/shortcomings.

Stage 3. Create /update a list that identifies the current company automation technology stack and industry alignment.

Stage 4. Create questionnaires.

Each of these stages can be longer or shorter depending on several factors, such as the following:

- The state of documentation, such as whether it is up to date and accurately describes the environment
- The size of the organization
- Whether this is a first assessment or whether an assessment has already taken place
- The availability of key stakeholders

Not all companies follow the same stages in creating an assessment, but the ones outlined here typically work well. To start, consider whether you already have automation in place. If the answer is no, you can skip the stage of creating an inventory as you won't have anything to add there. Industry

alignment is still relevant, as it helps you understand if your industry uses automation when you do not. Identifying needs and shortcomings is always relevant, as there are no perfect solutions.

If you already have automation in place, you need to understand what the automation is.

The first stage, which focuses on creating/updating an inventory for network automation, is the most challenging stage.

You should start by identifying the individuals/teams in the company that are using automation for networking ends. When I target teams, I typically ask for a single person to be responsible for the questionnaire, in order to reduce duplicates.

When you know who is using automation for networking, you have several options: ask for documentation, conduct interviews, or send questionnaires. I find that using questionnaires works best, as you can request all the information you want—even information that might not be reflected in the documentation. In addition, working with questionnaires is less time-consuming than conducting interviews. Craft a questionnaire that captures the tools and data people are using, to what end they are using those tools and that data, how often they are using them, and any challenges they are facing. You should tailor this questionnaire to your specific needs. The following are some examples of questions you might ask:

- 1.** Are you using any automation/configuration management tools? If so, which ones?
- 2.** Do you have any data dependencies? If so, what are they?
- 3.** To what purpose do you use the previously mentioned automation/configuration tools?
- 4.** To what extent are you using those tools?
- 5.** Who owns those tools?
- 6.** How long have you been using those tools?
- 7.** Where are those tools hosted?
- 8.** Do you face any challenges with those tools that you might be able to overcome with a different toolset?

9. How many people in the IT organization are familiar with automation?

The following are example of answers I typically see:

1. Yes. We are using Ansible, Git, and Grafana.
2. We constantly gather metric data from our network devices. We use it to power our Grafana dashboards. We also retrieve the network devices' configurations every week.
3. We use Ansible for device configuration, Git to store configuration files, and Grafana for metric visualization.
4. We use Git to store all our devices configurations, but we do not make all changes using Ansible. Many of our change windows are still done manually.
5. We have an operations department that takes care of the maintenance of our tooling—that is, upgrading and patching.
6. We have fairly recently adopted Ansible and Git; we have been using them for about 6 months. Grafana is even newer; we've been using it about 3 months.
7. Because of regulatory requirements, we host all our tooling on premises.
8. We are still in the infancy of adoption, but so far the main challenge has been the lack of in-house knowledge of Grafana.
9. Not many. We are trying to grow that number. Right now, about 10%.

After crafting your questionnaire and forwarding it to the responsible individuals/teams, you will receive answers and need compile them all into a single knowledge base.

At this stage, you should already have a clear picture of what you have at a global level; this is different from the local picture that each team has. Your knowledge of the global picture will allow you to score each tool and technology used in terms of maturity level, internal skills available, and current investment. [Figure 7-3](#) shows a visual representation of an evaluation of Ansible as tool from a hypothetical questionnaire where 2 departments of 30 people each were actively using Ansible for their day-2 operational tasks out of the total 10 departments surveyed. The Ansible playbooks of 1 of the 2

departments consisted of complex end-to-end actions. There is some investment, but the company clearly has skills available if Ansible is the tool of choice because it could use some of those 60 people to upskill the others.



Figure 7-3 Technology Maturity Level Chart

Having a visual representation helps others understand where you are for a given technology. You can also create aggregate visualizations based on data from specific teams or even the whole organization.

In terms of network automation, I like to classify companies and departments according to four levels:

- 1. No automation:** This classification is self-explanatory: It means doing things manually.
- 2. Provisioning and configuration automation:** The organization is using automation tools such as Ansible or Terraform to provision, configure, and manage the network.
- 3. Orchestration management:** The organization has orchestrated end-

to-ends flows (for example, it might be able to deploy a new service using an orchestration pipeline such as Jenkins). However, the organization still relies on real configuration files.

4. Service abstractions: This stage is where the network is abstracted. Few organizations are at this stage. When the organization needs to deploy something, it simply change an abstracted representation of it, and the system maps that change to configuration in the required equipment. The organization doesn't manage specific configuration files; instead, it manages more generic representations of those files.

To help you understand these levels, [Examples 7-1](#), [7-2](#), and [7-3](#) illustrate the evolution between level 2, 3, and 4 for a VLAN configuration task.

[Example 7-1](#) shows an Ansible playbook that configures the VLAN highlighted by using Cisco's *ios_config* Ansible module. This is tightly coupled to the syntax required to configure a VLAN on a specific switch.

Example 7-1 *Deploying a New VLAN on a Network Switch by Using Ansible*

```
$ cat vlan_deployment.yml
---
- name: Ansible Playbook to configure an interface
  hosts: ios

  tasks:
    - name: configure interface
      cisco.ios.ios_config:
        lines:
          - name servers
          parents: vlan 101
```

[Example 7-2](#) goes a step further than [Example 7-1](#). It is no longer a manually triggered Ansible playbook that does a single action; it is a Jenkins pipeline that executes the previous playbook and also executes a second one to verify that the changes were correctly applied. However, this example still uses the previous Ansible playbook, which requires a user to know the exact syntax of the commands to apply to configure a new VLAN.

Example 7-2 Jenkins Pipeline for VLAN Configuration

```
pipeline {
    agent any

    stages {
        stage("Configure") {
            steps {
                sh 'ansible-playbook -i hosts vlan_deploy.yml'
            }
        }

        stage("Test") {
            steps {
                sh 'ansible-playbook -i hosts vlan_verify.yml'
            }
        }
    }
}
```

[Example 7-3](#) shows the same pipeline as [Example 7-2](#), but this example has a data model in place, where a user must only provide a VLAN number and name to configure on the network device. You achieve this through templating. A more advanced example could render different templates when connecting to different operating systems or different switches; refer to [Chapter 5, “Using Ansible for Network Automation,”](#) for more on how to build such templates.

Example 7-3 Deploying a New VLAN Using an Abstracted Input

```
$ cat vlan_deployment.yml
---
- name: Ansible Playbook to configure a vlan
  hosts: ios

  tasks:
    - name: import vars
```

```

include_vars: vlan.yml

- name: render a Jinja2 template onto an IOS device
  cisco.ios.ios_config:
    src: "{{ vlan.j2 }}"

$ cat vlan.j2
{% for item in vlan %}

vlan {{ item.id }}
  name {{ item.name }}

{% endfor %}

$ cat vlans.yml
vlans:
  - id: 101
    name: servers

```

Based on your global knowledge base and the previous examples, you should be able to place your organization in one of these levels. Knowing your current level is relevant for the next step in the process of creating an automation strategy, which is planning goals and objectives. You should align your goals and objectives to match your current level if you still have departments or tasks in a lower level or align them to achieve the next level. Try to evolve automation in a step-by-step fashion. For example, do not try to plan for Level 4 if you are currently at Level 1. You can visualize the levels as a flight of stairs, as illustrated in [Figure 7-4](#).

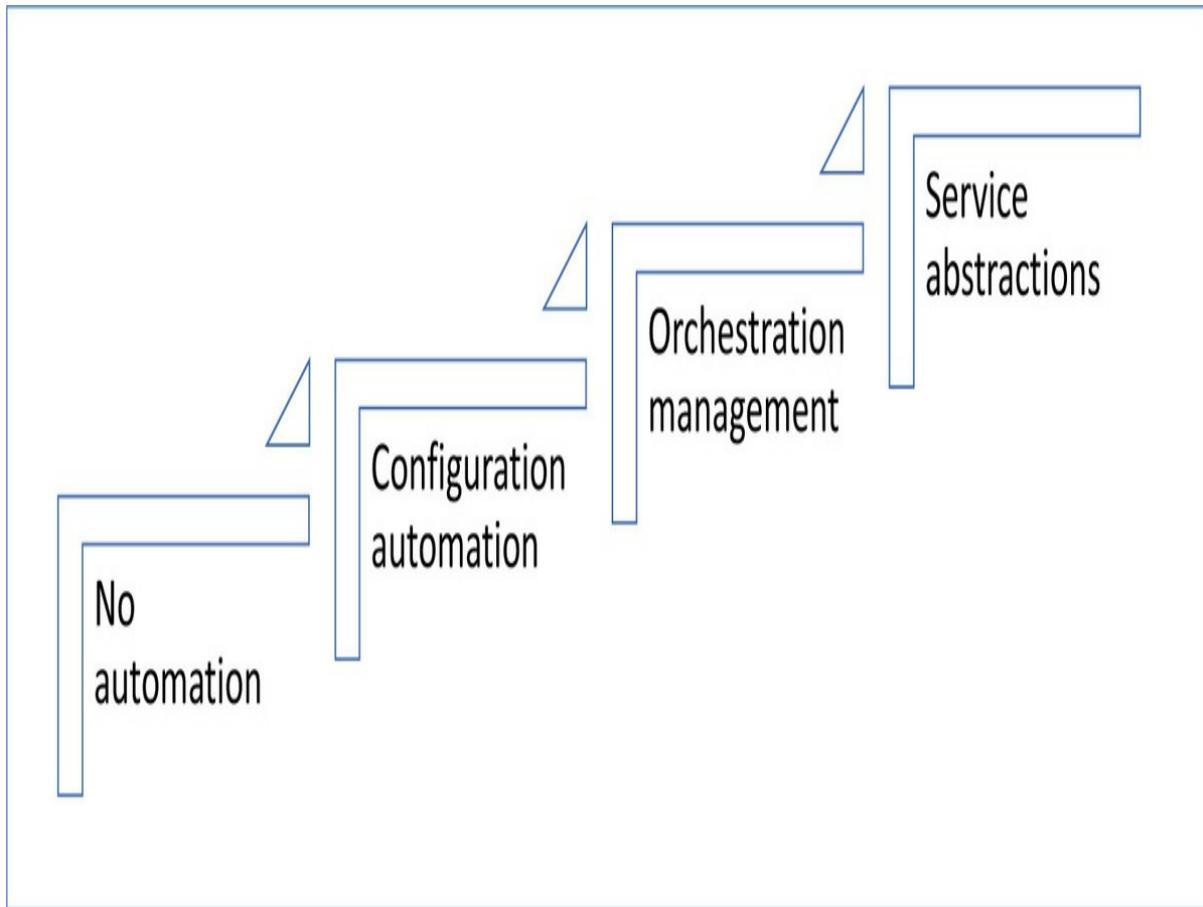


Figure 7-4 Enterprise Automation Levels

The last stage of the assessment effort is to compare your newly crafted knowledge base to the industry state. For this, you should research what other companies like yours are doing. Are they using Terraform to spin up infrastructure, or are they manually creating their virtual machines? This is especially important because you do not want to be behind the curve.

By the end of this assessment effort, you should have answers to all these questions:

- Do you have automation in place?
- What tools do you already have in place?
- Is your workforce trained or knowledgeable in automation?
- Is there need for automation?
- Are you ahead or behind the industry in terms of automation?

- Is there support for automation from stakeholders?

Knowing where you stand is paramount to developing a good automation strategy. Do not underestimate the assessment phase.

Culture and Skills

Although there is not a section of the automation strategy document for culture and skills, organizational culture and employees' skills are major factors that influence your strategy planning and possibly its success. Digital transformation is not only about technology but about people.

Recall from earlier in this chapter that one of the automation strategy pillars is skills. Networking as a technology is old, and automation is not a natural evolution of networking; it is a rather different vertical.

Many if not most network engineers do not have automation in their skillset. It is something that they have been adding, however, and you need to take this into consideration for your automation strategy. Does your workforce (which might be just you) have the skills required to achieve your goals? If yes, that's great! If not, there are two options:

- **Training and upskilling:** If you choose to go this route, you should add training and upskilling as tactics. Bear in mind that there are costs. Monetary costs are attached to trainings, certifications, and laboratories, and there are also time costs. People don't just learn new skills overnight, especially if those skills are unlike previously learned skills.
- **Hiring:** If you choose to go this route, again there will be both monetary and time costs.

Many companies today, big and small, are undertaking the transition from device-by-device networks to software-managed networks powered by automation. If that is happening in your organization, you are not alone. Whether to choose training an upskilling or hiring is a decision that needs to be made on a case-by-case basis. Both routes to obtain the skills of the future are valid.

Organizational culture also has an impact on the automation strategy. Depending on the domain, it may be easier to introduce automation. For

example, it is typically harder to introduce automation in networking than in software development because those practices are already embedded in the software culture. Unfortunately, the networking culture is typically very manual and dependent on key individuals. Adopting automation practices can be challenging in some companies where such culture is deeply rooted.

Some still also see automation as a job taker, even though it is more of a job changer. Failing to adapt may result in job losses, and many people fear automation. This can be a major challenge for companies and should not be underestimated as it can completely destroy your goals.

A term I have heard in the context of network changes is “culture of fear.” Some network folks are afraid to touch these ancient and critical network devices that can bring the entire network down in flames, and automating changes can seem really scary. This also applies to newer network components in the cloud. If adopting a new culture of NetDevOps, as described in [Chapter 6, “Network DevOps,”](#) is part of your strategy, you need to plan the transition from a “culture of fear” very carefully and include it in your automation strategy. If you think that the current culture needs to adapt in order to meet your goals, you must factor that in and plan accordingly.

Goals, Objectives, Strategies, and Tactics

With the knowledge of where you are as well as the current skill and culture of your organization culture in mind, you can start defining your automation strategy goals. To do so, you need to consider these questions:

- What do you want to achieve?
- What problems are you trying to solve?
- What is in need of improvement?

For ease of understanding, let’s say you are working for a telecommunications service provider, XYZ, that has 1000 employees, divided into several teams, including IT operations, software development, and support. These departments account for 400 out of the total 1000 employees at the company.

After an automation assessment consisting of a questionnaire to the relevant stakeholders, along with a compilation of the results under a single

knowledge base and comparison with the market, you have the results shown in [Table 7-1](#).

Table 7-1 XYZ Service Provider Assessment Knowledge Base

Tool	Ansible	Terraform	Python	Shell Scripting
Usage maturity (0–10)	5	2	2	7
Skills available (0–10)	8	5	3	1
Current investment (0–10)	8	2	2	1
Industry alignment (0–10)	6	8	4	1

Currently the company is using four different technologies for the automation of tasks. It seems that there is some lack of guidance in what to use. From your assessment, you could tell that shell scripting is the most mature of the four, probably because the company has been using it in operations for over 10 years, and the other tools have been adopted over time.

Among the three teams surveyed, Ansible is a clear winner in terms of skills available. Most of the engineers know Ansible even if their departments are not using it at the moment. Terraform seems to be gaining momentum, and the development team is well versed in it. In terms of the scripting languages, only the folks using them daily know how to use them.

The operations department Ansible is supported by Red Hat as they undertake critical tasks using it. This shows that there has already been a financial investment, unlike for any of the other tools the company is using.

Finally, in terms of industry alignment, although Ansible is still very relevant, some major players in the segment have realized benefits from using Terraform and are migrating toward that solution. Python and shell scripting

are not key technologies in this market segment.

From this assessment, you decide that the company is at Level 2, configuration automation. It does not have end-to-end pipelines, but it does have several tasks automated with the different technologies.

You can now define what you want to achieve, such as, “I want to accelerate the time to market of new services launched in my market segment.” This could be a goal. Do not forget to align the automation goals with your business strategy goals. You may have more than one goal, but you should typically have fewer than five.

Now, with these goals in mind, you must define objectives, which must be measurable and achievable. Here are two possible examples:

- Deploy a new VPN service from start to finish in under 24 hours.
- Start working on a new service request a maximum of 8 hours after the request is made.

Although it can be difficult to know what is achievable, you can consult subject matter experts in your company. If in doubt, aim for lower. Or have in mind that you were not sure what was achievable when evaluating the success of the strategy.

You will face many challenges when defining objectives. The two most common ones are defining too many objectives by not prioritizing the most important ones and lack of linkage to measurable KPIs.

During this type of exercise, companies typically come up with many objectives to reach the defined goals. Because an automation strategy is linked to a finite time frame, such as 1 year, it is important that objectives be achievable in the set time frame. This is where prioritization plays an important role.

For each of the objectives, you should try to quantify the cost to benefit—that is, how much money you need to invest and what benefit it will bring. Money is not necessarily the only metric; time is also a consideration, but often time can be translated to a dollar value, and hence cost is typically the metric used.

In terms of addressing the problem of linkage to measure KPIs, let us go back and use the methodology introduced earlier in this chapter and answer the seven questions applied to the two objectives:

- 1.** What is your desired outcome? Deploy a new VPN service from start to finish in under 24 hours.
- 2.** Why does this outcome matter for the company/department? If we achieve it, we would be faster than any of our competitors. This would allow us to attract new customers and retain current ones.
- 3.** How can you measure this outcome in an accurate and meaningful way? We can measure it by recording the time it takes from a customer requesting a service to when it is operational.
- 4.** How can people influence this outcome? Our teams can adopt network automation techniques to decrease the service deployment times. On the other hand, because it is currently a manual process, employees can register new service requests later than the actual request date.
- 5.** Can you tell if you have achieved the desired outcome? If so, how? We can tell if we have achieved it if all our services are within the 24-hour threshold.
- 6.** How often will you review the progress made? We aim to review it on a biweekly basis to understand the progress made.
- 7.** Who is ultimately responsible for this outcome? The operations department.

In this case, a KPI could be *time to deploy a new service*. You could also have KPIs to track each process within a new service deployment to better understand which of the processes are slowest. This would be a more granular approach.

It is important to note that people could influence this KPI to their benefit by registering the service later than when it was actually requested. Knowing this, if you choose this KPI, you should implement something to mitigate it as it could make your KPI measurements meaningless. It is very important to identify flaws of a KPI when defining it.

You can repeat this process for the second objective:

- 1.** What is your desired outcome? Start working on a new service request a maximum of 8 hours after the request is made.
- 2.** Why does this outcome matter for the company/department? Starting

to work on requests right away would help the company seem more responsive to customers and hence improve customer experience.

3. How can you measure this outcome in an accurate and meaningful way? We can measure it by recording the time it takes from when a customer initiates a new service request to when one of our engineers starts working on its fulfillment.
4. How can people influence this outcome? People can impact this by accepting the requests as early as possible.
5. Can you tell if you have achieved the desired outcome? If so, how? We can tell if we have achieved it if all our service requests are accepted below the 8-hour threshold.
6. How often will you review the progress made? We aim to review it on a biweekly basis to understand the progress made.
7. Who is ultimately responsible for this outcome? The operations department.

In this case, a KPI could be *time to accept a new service request*. In the event that there are many services request types that depend on different teams, you could define a KPI per service request type to measure in a more granular way.

The result of applying the technique is two objectives-linked measurable KPIs relevant to the company that can be tracked across a defined period of time.

After defining your goal and objectives, it is time to draft strategies to reach your goal. In this case, the goals are directly linked to objectives, but sometimes they span different objectives:

- (Objective) Deploy a new service from start to finish in under 24 hours.
 - (Strategy) Adopt automation processes for service deployment.
 - (Strategy) Adopt orchestration processes for service deployment.
- (Objective) Start working on a new service request a maximum of 8 hours after the request is made.
 - (Strategy) Create a new platform for service requests that is always

up.

- (Strategy) Improve the notification system for new service requests.

These strategies are paths to the goal. Put some thought into strategies, as it is important that they lead to the goal. Through prioritization, you can select the strategies that are most effective.

The last step in this process is to create tactics, which are the low-level, specific ways of fulfilling the strategies:

- (Strategy) Adopt automation processes for service deployment.
 - (Tactic) Use Ansible to deploy configurations.
 - (Tactic) Use Jinja2 to generate configurations.
 - (Tactic) Use virtual devices for testing prior to deploying configurations to production.
- (Strategy) Adopt orchestration processes for service deployment.
 - (Tactic) Develop an in-house orchestrator to interact with the current software ecosystem.
 - (Tactic) Educate the operations department on the new orchestrator system.
- (Strategy) Create a new platform for service requests that is always up.
 - (Tactic) Use a cloud SaaS for the new platform.
 - (Tactic) Use a microservices architecture for the new platform so that all components are decoupled and resilient.
- (Strategy) Improve the notification system for new service requests.
 - (Tactic) Replace the pull notification method with a push one.
 - (Tactic) Implement a mandatory acknowledgment from the receiver when a message is sent.

In the tactics is where you define what tools to use. Remember that the automation strategy document is for the whole organization. It is very important to understand the scope of your strategies and circle back to what you have learned from your assessment.

Some of the following questions can help you choose the *right tool for you*, which in some cases may be different from the right tool for the job:

- Is it router configuration automation or another type of system?
- Is it cloud provisioning or on premises?
- Do we have in-house knowledge, or will we need to procure it?
- Do we have rack space in the data center for a new solution?
- Do we need vendor support?
- Is the technology aligned with industry best practices?
- Is it scalable enough for all our use cases?

You need to do this exercise for all your strategies. This exercise takes time and effort, but the reward is amazing.

Note

Remember that a strategy should have a time frame associated with it. Aim for 6 months to 2 years for a strategy; after that time, revisit the strategy and, if needed, update it.

Finally, it is important not to forget to consult other strategy documents and add excerpts from it to the automation strategy if they are relevant. For example, say that you have defined a strategy to use a cloud SaaS for your future service request platform. Service requests more often than not contain personal identifiable information (PII), among other data. It is important to verify that your data strategy supports having this type of data in the cloud. Furthermore, your systems should comply with any strategies defined in that document. Say that your data strategy has the following objectives and strategies to fulfill your goal:

- (Objective) Centralize all data under a single umbrella.
 - (Strategy) Adopt common storage processes for common types of data.
 - (Strategy) Adopt common processing methods for common types of

data.

Expanding the strategies, you come up with the following tactics:

- (Strategy) Adopt common storage processes for common types of data.
 - (Tactic) Use AWS S3 to store all file data.
 - (Tactic) Use AWS Redshift to store all analytical data.
 - (Tactic) Use AWS DynamoDB to store all metadata used.
 - (Tactic) All data must be encrypted at rest.
- (Strategy) Adopt common processing methods for common types of data.
 - (Tactic) Before storage, analytical data must pass by the EMR ETL engine for aggregation.
 - (Tactic) All PII data must be anonymized using an in-house-developed system.

Some of these tactics directly impact your automation strategy. Any system you intend to develop must comply with these strategies. For example, your cloud SaaS platform will have to make use of the anonymization system before you store any service request PII data. For this reason, it is good to have an excerpt of it in your automation strategy document, so stakeholders can see why you are following this approach without having to jump from one document to another.

The same logic applies for the financial strategy and other strategy documents: You should consult them and add any excerpts that are relevant for your context.

ABD Case Study

The previous walkthrough has shown you what an automation strategy can look like. To make the concepts even clearer, let's look at a real example of the network automation part of an automation strategy for a big company (with more than 1000 employees) called ABD, where in the recent past the network experienced production-level issues that had business impact.

The goal was clear for the stakeholders involved even before an assessment

was made: The network must be able to support the needs of the business and stop being a bottleneck.

With the goal set, an extensive assessment was initiated to understand where the company was in terms of network automation and also to understand what issues existed with the current network. This type of assessment is typically beyond the scope of an automation strategy as the focus should only be on the automation piece. However, as a result of outages, understanding the “where we are” part may extend beyond automation.

The assessment was done by an independent third party, but it involved multiple internal teams, including the networking team and the operations teams. It consisted of interviews and questionnaires to capture the current situation. The following is a summary of the results:

- The network is running on legacy software versions.
- The current network is experiencing a large number of hardware failure occurrences.
- The network is monitored using a legacy system that is not configured for notifications.
- Network maintenance and proactive changes are manually executed.
- Network rollbacks are manually executed.
- There are no automation tools in use in production.
- The networking team uses Ansible in the lab environment.
- The operations department is skilled with Python.
- The networking department is skilled with Ansible.

ABD did not have much to assess from a network automation perspective as the company was barely using automation at all. However, it is still a good idea to evaluate ABD in terms of the four criteria and put less emphasis on each individual tool; instead have a generic indicator for the whole thing, see the Organization column in [Table 7-2](#). In the table you can see that barely any investment has been made in automation; ABD has invested just a little in Ansible for the lab environment. In terms of skills, the workforce is knowledgeable in two tools, Python and Ansible, but this knowledge is divided between two different teams, networking and operations. The

maturity score reflects the lack of automation solutions deployed. Finally, this organization is not aligned with its industry segment as other companies in the same segment are using automation. You see high scores for some specific tools because those individual tools are aligned with the industry for their use case.

Table 7-2 Technology Assessment Knowledge Base

Tool	Ansible	Python	Organization
Usage maturity (0–10)	2	0	1
Skills available (0–10)	6	5	6
Current investment (0–10)	2	0	1
Industry alignment (0–10)	8	6	1

With the “where we are” picture, the responsible stakeholders defined objectives to address their goal:

- Changes to the network must be made in under 1 hour and without major impact.
- The network must have an uptime of 99%.
- The network must support the addition of two new services every month.

Note

For this case, we are only looking at only the network automation objectives.

For each of these objectives, KPIs had to be created. By using the seven-question method, the company defined the following KPIs:

- Number of new services deployed per month
- Time taken, in seconds, for each step of a change window
- Duration of services affected during a change window

- Number of services affected during a change window
 - Number of service-affecting network outages per month
 - Duration of service-affecting network outages per month
-

Note

Modern KPIs can also be focused on customer experience rather than infrastructure metrics, such as results from a customer feedback form.

This was the outcome of one of the exercises for the KPI definition, which was repeated for every objective:

1. What is your desired outcome? Changes to the network must be made in under 1 hour and without major impact.
2. Why does this outcome matter for the company/department? We cannot have the network impact business operations. The longer the maintenance windows are, the longer potentially other teams cannot do their functions. The network is a foundation for everything else.
3. How can you measure this outcome in an accurate and meaningful way? We can measure the time the maintenance windows take, and we can measure which services were affected and for how long.
4. How can people influence this outcome? The team can automate the processes it uses to make changes, aligning to what the industry is already doing and improving the consistency and speed of the network changes.
5. Can you tell if you have achieved the desired outcome? If so, how? We can tell by verifying whether the time the windows take is decreasing, along with the number of service outages.
6. How often will you review the progress made? It should be reviewed on a quarterly basis.
7. Who is ultimately responsible for this outcome? The networking team.

By this stage, the company knew what it wanted to achieve. It was time to

determine how they would achieve it. They defined the following strategies:

- Improve the current network monitoring system.
- Adopt automation tools/processes to perform network changes during planned windows.
- Create automated failover processes that occur during network outages.

Finally, the company detailed each of these strategies into tactics, using the knowledge gathered from the assessment related to the workforce skillset:

- (Strategy) Improve the current network monitoring system.
 - (Tactic) Collect new metrics from the equipment (CPU, memory, and interface drops).
 - (Tactic) Configure automated alarms on the monitoring system.
 - (Tactic) Use past log information to train an ML model of predictive maintenance.
 - (Tactic) Configure monitoring for software versioning.
- (Strategy) Adopt automation tools/processes to perform network changes during planned windows.
 - (Tactic) Use Ansible playbooks for production changes.
 - (Tactic) Use Ansible playbooks for automatic rollbacks.
 - (Tactic) Use Git to store the playbooks and configurations.
 - (Tactic) Verify service-affecting changes in a virtual lab before applying them to the production environment.
 - (Tactic) Automatically verify the status of the production network before and after a change.
- (Strategy) Create automated failover processes that occur during network outages.
 - (Tactic) Use Python to automate traffic rerouting in the event of byzantine hardware failure.
 - (Tactic) Use Python to automate the tasks required for failover in the event of a complete data center outage (for example, DNS

entries).

- (Tactic) Use past log information to train an ML model of predictive maintenance.

Another section of this automation strategy was a linkage to the data strategy. This was justified by the specific mentions of how to store data in that strategy, which directly impacts the folks implementing the automation strategy. The following objective and strategies were transcribed from that document:

- (Objective) Have full control over stored data.
 - (Strategy) Store all data on company premises.
 - (Strategy) Adopt a controlled access mechanism for data.

These strategies were also specified in the form of tactics. For the scope of the automation strategy, they were also relevant and therefore transcribed:

- (Strategy) Store all data on company premises.
 - (Tactic) Log data must be moved to San Francisco's data center Elasticsearch.
 - (Tactic) Metric data must be moved to San Francisco's data center InfluxDB.
- (Tactic) After 3 months, historical data must be moved to cold storage.
- (Strategy) Adopt a controlled access mechanism for data.
 - (Tactic) Active Directory should be used to authenticate and authorize any read/write access to data.
- (Tactic) An immutable accounting log should exist for any data storage solution.

You may see the importance of transcribing the data strategies and tactics. If the company did not do so, it might overlook the need to store the new collected device metrics in the proper place—in this case, San Francisco's data center. Furthermore, it is important for the company to investigate and plan how to build the machine learning predictive system if the historical data is moved to a different location as the move might add extra cost and complexity.

This was the overall structure of ABD's automation strategy, which was quite extreme in terms of the changes to the company culture—going from a completely manual operation model to introducing automation and eventually having an automated environment. Based on the maturity level and the changes desired, ABD set a 2-year goal for implementing this strategy, and it planned to revisit it every 3 months to monitor progress.

How to Use an Automation Strategy

After you have built your own automation strategy, what happens next? Well, you will need to use it. The first step in using your strategy is to build a planning document. This document should have a time span of a year or less, and it should highlight the project's milestones.

To create an action plan, you need to prioritize your tactics. Some companies like to focus on quick wins, and others prefer to start with harder and long-lasting projects first. There is not a single best way to prioritize. I find that having wins helps collaborators stay motivated, and because of that, I like to have in parallel long-lasting projects and smaller, more manageable ones.

Start with a tactic that you have identified as a priority and involve technical colleagues if you need the support. Break down the tactic into specific actions and assign dates for those actions. Let us illustrate with an example. Say that you are collecting new metrics from your equipment (CPU and memory). In order to achieve this tactic, a first step could be verifying whether the current monitoring system supports collecting such metrics or whether you need to acquire a monitoring system because you don't have one.

A second step could be configuring the monitoring system to collect these new metrics on one device. This would be called a proof of concept (PoC). If the PoC is successful, a third step could be configuring the monitoring system to collect these new metrics on 25% of the installed base. And a fourth step could be configuring it to collect from 50% of the installed base. A fifth and last step could be configuring the remaining 50%. Each of those actions should have a date assigned, along with an owner responsible for planning and execution. This owner could be a team rather than a single individual.

This type of phased approach is the most common and is typically the most successful. Jumping straight into a full-blown global system is difficult and

error prone. Try to follow a phased approach when defining a plan for a tactic unless it is a tactic with a small scope.

An action plan should include detailed steps for all the tactics that will happen in a particular period of time.

[Table 7-3](#) shows a formatted action plan for a single tactic. You can add different fields. Typically the minimum required fields are the owner and date fields. For the dates, you can be more specific than specifying quarters, as in [Table 7-3](#). A description column gives you a place to describe more complex actions that may not be fully understood based only on their names. Lastly, it is common to have a reference column, represented as Ref in our table, this simply provides a reader an easy way to refer to any specific row. We are using a numerical reference, but alpha-numeric is also commonly seen.

Table 7-3 *Action Plan for Adoption of Ansible and Creation of Playbooks for Change Windows*

Ref	Action	Owner	Target Start Date	Target End Date	Description
1	Investigate Ansible licensing	Licensing department	Q1	Q1	After the investigation, acquiring any license is also included, if needed.
2	Install Ansible in central virtual machine	IT department	Q1	Q1	
3	Provision user accounts to the virtual machine	IT department	Q1	Q1	
4	Identify activities of change windows	Operations department	Q1	Q1	
5	Create Ansible inventory file	Automation department	Q1	Q1	This should only include devices that will be affected by a future change window.
6	Create playbooks to automate identified activities	Automation department	Q2	Q4	Every playbook should be uploaded to a Git repository. Playbooks should be prioritized based on the activity priority in a change window.
7	Distribute VM access information to relevant stakeholders	IT department	Q2	Q2	Distribute information by email.
8	Test in one production device	Automation department	Q4	Q4	This assumes that playbooks have been tested previously in a virtualized staging environment

You can aggregate several tactics under the same action plan, but it may become very big and hard to read. Sometimes it is a good idea to use a different table for each tactic. A downside of this approach is that the delivery dates are spread out throughout the document.

In addition, visual representation of actions plans can help a lot in understanding and correlating all the time tables. You can see an example in [Figure 7-5](#), which shows a scaled-down chart version of [Table 7-3](#) with five tasks. You can see here that the first task is a predecessor for Tasks 2, 3, and 4, which may take place in parallel, and the fifth task is dependent on the three previous tasks. These dependencies are highlighted by the arrows. Furthermore, you can see on this chart who is responsible for each task.

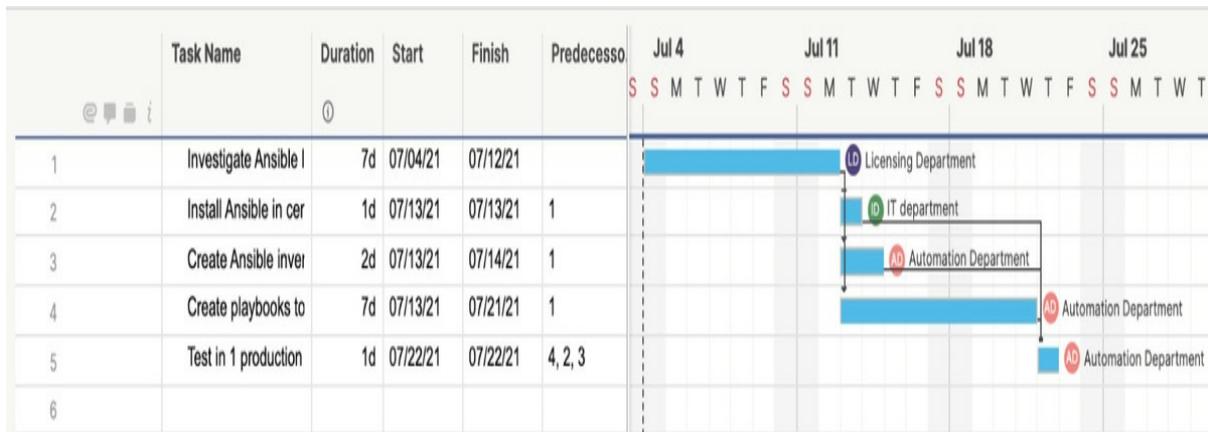


Figure 7-5 Gantt Chart of Tasks

Meanwhile, you should be monitoring your progress and KPIs. That is why you defined them in the first place, and you need to make sure they are being met. If they are not being met, you need to understand why and adjust, which may mean updating your automation strategy; as mentioned earlier, it is a living document, so adjustment is expected. However, keep in mind that you should not be changing your automation strategy every day. If you are making changes very frequently, try to understand the underlying reason: Are you defining unrealistic objectives? Are your company skills not fit for the goals?

Note

Incorporating a yearly cycle to assess the need for readjustment is typically helpful.

In summary, to fully use your automation strategy document, you need to create a project plans for each tactic where you define granular actions to be taken and highlight time lines and ownership. Implement your tactics according to your plan while measuring and monitoring your KPIs.

Summary

This chapter covers automation strategies. Just as a company would be without a business strategy, you should not have automation without an automation strategy. You should not wander around automating one-off actions without understanding what you really want to achieve and how doing so can be beneficial in the long term.

This chapter mentions the five key pillars that support an automation strategy:

- Business alignment
- Technology adoption
- Operating model
- Roadmap
- Skills

This chapter also covers the sections of an automation strategy document, as well as core components, such as KPIs, that make up the document's body.

Finally, this chapter guides you on building your own automation strategy document and provides network automation examples, tips on what to keep in mind during this journey, and how to translate the strategy into reality.

Review Questions

You can find answers to these questions in [Appendix A, “Answers to Review Questions.”](#)

- 1.** Which of the following is *not* a pillar for an automation strategy?
 - a.** Skills
 - b.** Culture
 - c.** Technology adoption
 - d.** Business alignment
- 2.** In an automation strategy, what are goals?
 - a.** Specific outcomes that are measurable, realistic, and achievable
 - b.** Metrics to evaluate where you are
 - c.** Specific actions to follow
 - d.** High-level purposes
- 3.** Which of the following fields is *not* mandatory in an action plan?
 - a.** Owner
 - b.** Action
 - c.** Target date
 - d.** Description
- 4.** What is the typical time range that an automation strategy addresses?
 - a.** 1 month
 - b.** 1 to 6 months
 - c.** 6 months to 2 years
 - d.** 5 years
- 5.** True or false: The automation strategy document should be crafted by only upper management stakeholders.

- a.** True.
 - b.** False.
- 6.** In an automation strategy document, can a strategy span multiple objectives?
 - a.** Yes, a strategy can span multiple objectives.
 - b.** No, a strategy cannot span multiple objectives.
- 7.** True or false: You cannot have other strategy documents linked to your automation strategy document.
 - a.** True
 - b.** False
- 8.** Which automation level is a company in if it has systems in place that abstract service complexity (that is, service abstraction)?
 - a.** 1
 - b.** 2
 - c.** 3
 - d.** 4
- 9.** In which part of the automation strategy document are the technology tools to be used enumerated?
 - a.** Goals
 - b.** Objectives
 - c.** Strategies
 - d.** Tactics
- 10.** How many tactics can a strategy have in an automation strategy document?
 - a.** 1–5

b. 5–10

c. 10–100

d. Unlimited

Appendix A. Answers to Review Questions

Chapter 1

1. B, Task based. In this scenario, you are triggering a task to change the hostnames of all your devices. Even though you are using the device's IP address to derive its location, this is not what triggers the automation.
2. B, Configuration. The intent is to configure new hostnames.
3. B, Pull model. Your network is already bandwidth constrained, and you don't want to stress it even further by having devices constantly pushing their metric data.
4. A, Push model. SNMP traps are sent by the devices to a configured destination.
5. Configuration drift means that devices have different configurations from each other or from your template configurations. It occurs when changes are made to the devices in an untracked manner (for example, during maintenance windows or tests). There are many ways to address this issue, and a popular one is to monitor device configurations compared against templates. If a change is detected, the known good configuration should be applied.
6. B, Ansible, and D, Python. Splunk and Kibana are visualization tools and cannot fulfill a configuration use case. In a multivendor environment, Cisco DNAc is not suitable due to its focus on Cisco-only equipment. Terraform's main use case is provisioning rather than configuration.
7. D, Terraform, and E, Python. Splunk is a visualization tool and cannot fulfill a provisioning use case. Cloud event-driven functions can be executed in response to events; they are not used to provision

infrastructure as a workflow. DNAC does not have cloud provisioning capabilities.

- 8.** D, Chef, and E, Ansible. Kibana is a visualization tool. Terraform is best suited for provisioning use cases, but in this case, you want to manage already provisioned servers. Although DNAC can help you manage your infrastructure, it is targeted at network infrastructure, not servers.
- 9.** A, Kibana, and C, Splunk. Kibana is an open-source log visualization tool that perfectly fits this use case. Grafana is also a visualization tool, but its focus is on metric data. Splunk is an enterprise solution that can consume logs in a central place. DNAC can show you logs from the managed devices, but this question does not say that all devices are Cisco and DNA supported; therefore, DNAC is not an ideal choice.
- 10.** B, Plan. Terraform generates an execution plan but does not execute it. It is a nice-to-have step to stage changes before actually applying them, but this step is not mandatory.
- 11.** C, Ansible. The security requirement that bans installation of an agent excludes Chef, as it is an agent-based solution. Terraform is focused on provisioning rather than configuring. As there is no mention of the devices being Cisco DNAC supported devices, the best answer is Ansible.
- 12.** B, False. Kibana excels at utilizing log data.
- 13.** A, True. Although their name has the keyword *cloud*, event-driven functions can interact with on-premises components as long as there is IP reachability.

Chapter 2

- 1.** A, Container list. All the other options are valid YANG components, according to RFC 6020.
- 2.** B, False. Indentation is key, unlike in other data formats. In a file, the number of indentation spaces must always be the same. By convention, two spaces are typically used.

3. A, scalar, and B, dictionary. In YAML, there are three data types: scalar, list, and dictionary.
4. A, Yes. The YANG shown here conforms to the specifications.
5. A, True. The payload starts and ends with `<rpc>` tags. This is because the whole process is based on remote procedure calls (RPCs).
6. B, PATCH. You should use PATCH if you only want to update a single field out of many. It allows you to keep every other previously configured value unchanged.
7. B, XML, and D, JSON. RESTCONF can be encoded using either XML or JSON.
8. B, False. Syslog can have slight differences, depending on the vendor implementation.
9. C, Log exporter. SSH and NETCONF need something on top in order to achieve automation; they are only protocols. Python can certainly achieve the goal, but the effort and complexity in comparison to log exporters is higher.
10. A, `curl`, and B, Postman. `curl` is a command-line utility tool for getting and sending data using URLs. Postman is a tool with a graphical interface that allows to interact with APIs.

Chapter 3

1. A, `\d`. This wildcard matches one digit.
2. C, `\s`. This wildcard matches a whitespace character.
3. A, 255.257.255.255, and B, 10.00.0.1. This is a relatively simple regex that will match more than valid IP strings as you can see from option A. To match only on valid IP addresses, you should use more complex regex expressions than the one shown.
4. B, Data aggregation. If your network is already constrained, you want to stress it as little as possible while obtaining the necessary monitoring metrics. Aggregating metrics in time ranges or using similar techniques

helps you achieve this.

5. B, Line chart. For CPU utilization, the history is important, and a line chart gives the viewer a good understanding of its evolution through time.
6. C, Gauge. A gauge is ideal for quickly understanding a metric when historical data does not add value—as is the case with memory utilization.
7. B, False. Alarms can trigger both manual human interventions and automatic actions, such as Ansible playbook executions or scripts.
8. A, Use AutoML. If you lack domain expertise in artificial intelligence or machine learning but are looking to create models, the best way is to take advantage of AutoML, which enables you to only provide the data.
9. A, One or more occurrences. The wildcard matches one or more occurrences of the previous character.
10. A, Regression. You are trying to predict a numeric value that the machine learning needs to process transactions.

Chapter 4

1. B, Python. Ansible is an open-source software tool written in Python.
2. A, YAML, and B,INI. Ansible inventory files can be written in three formats: YANG, JSON, or INI.
3. D, switch01.example.com. 10.20.20.1 is part of the hosts group *Spain*, and 10.10.10.1 and switch01.example.com are part of the group *Portugal*.
4. B, Host variable. The status is under host 10.10.10.1 and has the value “prod”.
5. B, In a separate file that is uploaded to the repository. The hosts variable folder is a fine place to have some variables if they are static to the hosts. Playbook variables are typically different from host variables, and you should separate your variables from your playbooks to enhance

reusability. Passing in variables during CLI execution is acceptable but not friendly for automation.

6. A, `{{ example }}`. The syntax for referring to a variable in Ansible is `{{ variable_name }}`.
7. Ansible Vault allows you to store variables in encrypted format. When you have sensitive variables, you should use a Vault instead of saving your secrets in plaintext.
8. B, plays. A playbook is composed of one or more plays in a sequence.
9. B, False. By default, Ansible runs each task on all hosts affected by a play before starting the next task on any host.
10. B, False. In use cases that require asynchronous communications (for example, long-running tasks), you can use the `async` and `poll` keywords to achieve this type of behavior.
11. C, `until`. `until` retries a task until a certain condition is met. This is particularly useful for verifications that may not succeed at first try.
12. B, `-f`. By default Ansible's number of forks is 5. If you have the processing power available and want to use more forks, you can configure this by using the `-f` option.

Chapter 5

1. B, False. The `lookup` function only works for local files.
2. A, Base-64. You must decode the output before you can read it.
3. B, `delegate_to`. By using the `delegate_to` function, you can execute the task in the host you specify; in this case, you would specify localhost.
4. A, Yes. `ansible_facts` captures the interfaces' IP addresses.
5. A, `command`. There is no of using the package manager **yum**. The machines are Linux servers, not Cisco IOS devices, so `ios_config` is also out. `slurp` is a remote file module.
6. A, `os_family`. The `os_family` variable takes as a value the operating

system of the host you connect to if you are gathering facts. Based on this, you can choose the appropriate installation method.

7. B, No, you can still use Ansible. When no module is available—which can happen with in-house software—you can use the custom API module or build your own Ansible module.
8. C, Use the *lookup* function. Although the *uri* module would also work, the preferred way to crawl websites is by using the *lookup* function.
9. D, *netconf_get*. Using this module is the easiest way to retrieve information using NETCONF.
10. B, *absent*. The *absent* state deletes the virtual machine with the given name if it exists.

Chapter 6

1. B, False. NetDevOps is the adoption of DevOps software practices in networking. It is not a software development methodology.
2. A, Configuration drift. Infrastructure as code helps reduce configuration drift, as do NetDevOps CI/CD practices, by eradicating ad hoc changes in network devices.
3. C, Create. The DevOps phases are plan, code, build, test, release, deploy, operate, and monitor.
4. A, Declarative pipelines, and C, Scripted pipelines. Scripted and declarative are the two supported pipeline types supported by Jenkins.
5. A, Jenkins. Ansible and Terraform are configuration management/provisioning tools, and Git is a source control system.
6. A, Distributed. Git has distributed architecture, keeping full copies in each of the clients' machines.
7. A, True. Jenkins uses a controller/agent architecture, and you can easily scale it by simply adding more agents.
8. A, **git branch**. This command creates a new parallel and independent

branch that is initially just like the one it branches out of.

9. A, **git checkout**. This command followed by the commit hash points the head to that commit and modifies the repository to reflect its state at the time of the commit.
10. D, Unlimited. You can define any number of steps (instructions) within a single stage. However, it is not common to use many; rather, different commands are used in different stages.

Chapter 7

1. B, Culture. Although your current corporate culture is very important to consider when crafting an automation strategy document, it is not one of the five pillars.
2. D, High-level purposes. A goal in the context of an automation strategy is a high-level purpose that you want to achieve. For example, Company XYZ's goal might be to increase its profit margin.
3. D, Description. An action plan should have at least dates and owners for each action. However, you can add more fields if deemed necessary to make the plan more readable and understandable.
4. C, 6 months to 2 years. An automation strategy is bounded by a time frame, typically no less than 6 months and no more than 2 years, depending on the company size. After that time frame, the strategy should be revised.
5. B, False. The automation strategy document should be an effort between a wide variety of stakeholders, from technical to upper management. It is a companywide effort.
6. A, Yes, a strategy can span multiple objectives. There are cases in which a strategy contributes to more than one objective.
7. B, False. You can and should link other strategy documents if they are relevant in the automation context. Examples of commonly linked documents are the financial strategy and the data strategy.

- 8.** D, 4. The four levels are 1: No automation at all, 2: Configuration automation 3: Orchestration management, 4: Service abstraction.
- 9.** D, Tactics. Tactics are the low-level steps to achieve strategies, and you use them to enumerate the tools to be used.
- 10.** D, Unlimited. There is no limit to how many tactics you can define to achieve a specific strategy.