

Steven Armstrong

DevOps for Networking

Boost your organization's growth by incorporating networking in the DevOps culture



Packt>

DevOps for Networking

Boost your organization's growth by incorporating networking in the DevOps culture

Steven Armstrong

Packt

BIRMINGHAM - MUMBAI

DevOps for Networking

Copyright © 2016 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: October 2016

Production reference: 1261016

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78646-485-9

www.packtpub.com

Credits

Author

Steven Armstrong

Project Coordinator

Judie Jose

Reviewer

Daniel Jonathan Valik

Proofreader

Safis Editing

Commissioning Editor

Pratik Shah

Indexer

Pratik Shirodkar

Acquisition Editor

Namrata Patil

Graphics

Kirk D'Penha

Content Development Editor

Abhishek Jadhav

Production Coordinator

Shantanu N. Zagade

Technical Editor

Mohd Riyan Khan

Cover Work

Shantanu N. Zagade

Copy Editor

Dipti Mankame

About the Author

Steven Armstrong is a DevOps solution architect, a process automation specialist, and an honors graduate in Computer and Electronic Systems (BEng) from Strathclyde University in Glasgow.

He has a proven track record of streamlining company's development architecture and processes so that they can deliver software at pace. Specializing in agile, continuous integration, infrastructure as code, networking as code, Continuous Delivery, and deployment, he has worked for 10 years for leading consulting, financial services, benefits and gambling companies in the IT sector to date.

After graduating, Steven started his career at Accenture Technology solutions as part of the Development Control Services graduate scheme, where he worked for 4 years, then as a configuration management architect helping Accenture's clients automate their build and deployment processes for Siebel, SAP, WebSphere, Weblogic, and Oracle B2B applications.

During his time at Accenture, he worked within the development control services group working for clients, such as the Norwegian Government, EDF Energy, Bord Gais, and SABMiller. The EDF Energy implementation led by Steven won awards for "best project industrialization" and "best use of Accenture shared services".

After leaving Accenture, Steven moved on to the financial services company, Cofunds, where he spent 2 years creating continuous integration and Continuous Delivery processes for .Net applications and Microsoft SQL databases to help deploy the financial services platform.

After leaving Cofunds, Steven moved on to Thomsons Online Benefits, where he helped create a new DevOps function for the company. Steven also played an integral part in architecting a new private cloud solution to support Thomsons Online Benefits production applications and set up a Continuous Delivery process that allowed the Darwin benefits software to be deployed to the new private cloud platform within minutes.

Steven currently works as the technical lead for Paddy Power Betfair's i2 project, where he has led a team to create a new greenfield private cloud platform for Paddy Power Betfair. The implementation is based on OpenStack and Nuage VSP for software-defined networking, and the platform was set up to support Continuous Delivery of all Paddy Power Betfair applications. The i2 project implementation was a finalist for the OpenStack Super User Award and won a RedHat Innovation Award for Modernization.

Steven is an avid speaker at public events and has spoken at technology events across the world, such as DevSecCon London, OpenStack Meetup in Cluj, the OpenStack Summit in Austin, HP Discover London, and most recently gave a keynote at OpenStack Days Bristol.

Acknowledgments

I would most importantly like to thank my girlfriend Georgina Mason. I know I haven't been able to leave the house much at weekends for 3 months as I have been writing this book, so I know it couldn't have been much fun. But thank you for your patience and support, as well as all the tea and coffee you made for me to keep me awake during the late nights. Thank you for being an awesome girlfriend.

I would like to thank my parents, June and Martin, for always being there and keeping me on track when I was younger. I would probably have never got through university never mind written a book if it wasn't for your constant encouragement, so hopefully, you both know how much I appreciate everything you have done for me over the years.

I would like to thank Paddy Power Betfair for allowing me the opportunity to write this book and our CTO Paul Cutter to allow our team to create the i2 project solution and talk to the technology community about what we have achieved.

I would also like to thank Richard Haigh, my manager, for encouraging me to take on the book and all his support in my career since we started working together at Thomsons Online Benefits.

I would like to thank my team, the delivery enablement team at Paddy Power Betfair, for continually pushing the boundaries of what is possible with our solutions. You are the people who made the company a great innovative place to work.

I would like to thank all the great people I worked with throughout my career at Paddy Power Betfair, Thomsons Online Benefits, Cofunds, and Accenture, as without the opportunities I was given, I wouldn't have been able to pull in information from all those experiences to write this book.

I would also like to thank Nuage networks for permitting me to write about their software-defined networking solution in this book.

About the Reviewer

Daniel Jonathan Valik is an industry expert in cloud services, cloud native technologies, IOT, DevOps, infrastructure automation, containerization, virtualization, microservices, unified communications, collaborations technologies, Hosted PBX, telecommunications, WebRTC, unified messaging, Communications Enabled Business Process (CEBP) design, and Contact Center Technologies.

He has worked in several disciplines such as product management, product marketing, program management, evangelist, and strategic adviser for almost two decades in the industry.

He has lived and worked in Europe, South East Asia, and now in the US. Daniel is also an author of several books about cloud services, universal communications and collaborations technologies, which includes Microsoft, Cisco, Google, Avaya, and others.

He holds dual master's degrees: Master of Business Administration (MBA) and Master of Advanced Studies (MAS) in general business. He also has a number of technical certifications, including Microsoft Certified Trainer (MCT). For more information about Daniel, refer to his blogs, videos, and profile on LinkedIn (<https://www.linkedin.com/in/danielvalik>).

www.PacktPub.com

eBooks, discount offers, and more

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customer@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www.packtpub.com/mapt>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Table of Contents

Preface	ix
Chapter 1: The Impact of Cloud on Networking	1
An overview of cloud approaches	1
Public clouds	2
Private cloud	3
Hybrid cloud	4
Software-defined	4
The difference between Spanning Tree and Leaf-Spine networking	5
Spanning Tree Protocol	5
Leaf-Spine architecture	7
OVSDB	9
Changes that have occurred in networking with the introduction of public cloud	12
An overview of AWS	12
OpenStack overview	14
The AWS approach to networking	16
Amazon VPC	16
Amazon IP addressing	19
Amazon security groups	19
Amazon regions and availability zones	20
Amazon Elastic Load Balancing	20
The OpenStack approach to networking	21
OpenStack services	22
OpenStack tenants	23
OpenStack neutron	23

Provisioning OpenStack networks	24
OpenStack regions and availability zones	35
OpenStack instance provisioning workflow	36
OpenStack LBaaS	37
Summary	38
Chapter 2: The Emergence of Software-defined Networking	39
Why SDN solutions are necessary	39
How the Nuage SDN solution works	41
Integrating OpenStack with the Nuage VSP platform	44
Nuage or OpenStack managed networks	47
The Nuage VSP software-defined object model	49
Object model overview	49
How the Nuage VSP platform can support greenfield and brownfield projects	62
The Nuage VSP multicast support	68
Summary	71
Chapter 3: Bringing DevOps to Network Operations	73
Initiating a change in behavior	74
Reasons to implement DevOps	75
Reasons to implement DevOps for networking	77
Top-down DevOps initiatives for networking teams	79
Analyzing successful teams	79
Mapping out activity diagrams	81
Changing the network team's operational model	84
Changing the network team's behavior	86
Bottom-up DevOps initiatives for networking teams	88
Evangelizing DevOps in the networking team	88
Seeking sponsorship from a respected manager or engineer	90
Automating a complex problem with the networking team	91
Summary	93
Chapter 4: Configuring Network Devices Using Ansible	95
Network vendors' operating systems	96
Cisco Ios and Nxeos operating system	96
Juniper Junos operating system	98
Arista EOS operating system	98
Introduction to Ansible	99
Ansible directory structure	100
Ansible inventory	101
Ansible modules	102

Ansible roles	103
Ansible playbooks	104
Executing an Ansible playbook	106
Ansible var files and jinja2 templates	106
Prerequisites using Ansible to configure network devices	107
Ansible Galaxy	108
Ansible core modules available for network operations	110
The <code>_command</code> module	112
The <code>_config</code> module	113
The <code>_template</code> module	114
Configuration management processes to manage network devices	114
Desired state	115
Change requests	119
Self-service operations	119
Summary	121
Chapter 5: Orchestrating Load Balancers Using Ansible	123
Centralized and distributed load balancers	123
Centralized load balancing	125
Distributed load balancing	126
Popular load balancing solutions	126
Citrix NetScaler	127
F5 Big-IP	130
Avi Networks	132
Nginx	133
HAProxy	135
Load balancing immutable and static infrastructure	137
Static and immutable servers	138
Blue/green deployments	139
Using Ansible to Orchestrate load balancers	142
Delegation	142
Utilizing <code>serial</code> to control roll percentages	143
Dynamic inventories	147
Tagging metadata	147
Jinja2 filters	148
Creating Ansible networking modules	149
Summary	150

Chapter 6: Orchestrating SDN Controllers Using Ansible	153
Arguments against software-defined networking	154
Added network complexity	155
Lack of software-defined networking skills	156
Stateful firewalling to support regularity requirements	158
Why would organizations need software-defined networking?	159
Software-defined networking adds agility and precision	160
A good understanding of Continuous Delivery is key	161
Simplifying complex networks	162
Splitting up network operations	162
New responsibilities in API-driven networking	164
Overlay architecture setup	164
Self-service networking	171
Immutable networking	174
A/B immutable networking	174
The clean-up of redundant firewall rules	176
Application decommissioning	177
Using Ansible to orchestrate SDN controllers	178
Using SDN for disaster recovery	179
Storing A/B subnets and ACL rules in YAML files	181
Summary	183
Chapter 7: Using Continuous Integration Builds for Network Configuration	185
Continuous integration overview	186
Developer continuous integration	188
Database continuous integration	190
Tooling available for continuous integration	194
Source control management systems	194
Centralized SCM systems	195
Distributed SCM systems	196
Branching strategies	197
Continuous integration build servers	199
Network continuous integration	201
Network validation engines	203
Simple continuous integration builds for network devices	205
Configuring a simple Jenkins network CI build	206
Adding validations to network continuous integration builds	209
Continuous integration for network devices	210
Continuous integration builds for network orchestration	211
Summary	213

Chapter 8: Testing Network Changes	215
Testing overview	216
Unit testing	216
Component testing	217
Integration testing	217
System testing	218
Performance testing	219
User acceptance testing	220
Why is testing relevant to network teams?	221
Network changes and testing today	222
Quality assurance best practices	227
Creating testing feedback loops	230
Continuous integration testing	231
Gated builds on branches	233
Applying quality assurance best practices to networking	234
Assigning network testing to quality gates	236
Available test tools	238
Unit testing tools	238
Test Kitchen example using OpenStack	239
Network checklist	241
Network user journey	242
Quality of Service	243
Failover testing	244
Network code quality tooling	244
Summary	246
Chapter 9: Using Continuous Delivery Pipelines to Deploy Network Changes	249
Continuous integration package management	250
Continuous Delivery and deployment overview	254
Deployment methodologies	258
Pull model	258
Push model	260
When to choose pull or push	261
Packaging deployment artifacts	262
Deployment pipeline tooling	265
Artifact repositories	266
Artifactory	266
CD pipeline scheduler	268
Jenkins	269

Deploying network changes with deployment pipelines	273
Network self-service	273
Steps in a deployment pipeline	273
Incorporating configuration management tooling	275
Network teams' role in Continuous Delivery pipelines	276
Failing fast and feedback loops	276
Summary	277
Chapter 10: The Impact of Containers on Networking	279
Overview of containers	279
Solaris Zones	282
Linux namespaces	283
Linux control groups	285
Benefits of containers	285
Deploying containers	286
CoreOS	287
etcd	287
Docker	288
Docker registry	288
Docker daemon	288
Packaging containers	289
Dockerfile	289
Packer-Docker integration	289
Docker workflow	291
Default Docker networking	292
Docker user-defined bridge network	293
Docker Swarm	294
Docker machine	294
Docker Compose	295
Swarm architecture	296
Kubernetes	298
Kubernetes architecture	298
Impact of containers on networking	303
Summary	304
Chapter 11: Securing the Network	307
The evolution of network security and debunking myths	307
Account management	308
Network device configuration	310
Firewalling	310
Vulnerability detection	311
Network segmentation	312

Securing a software-defined network	314
Attacks at Overlay	315
Attacks on the underlay network?	316
Attacks on the SDN controller	318
Network security and Continuous Delivery	319
Application connectivity topology	320
Wrapping security checks into continuous integration	321
Using Cloud metadata	322
Summary	325
Index	327

Preface

The title of this book is "DevOps For Networking". DevOps, as you are probably well-aware, is an abbreviated amalgamation of "Development" and "Operations", so why does it have any significance to networking? It is true that there is no "Net" in the DevOps name, though it is fair to say that the remit of DevOps has extended well beyond its initial goal.

The initial DevOps movement sought to remove the "chucking it over the fence" and reactive mentality that existed between development and operations teams, but DevOps can be efficiently used to promote collaboration between all teams in IT, not just Development and Operations staff.

DevOps, as a concept, initially aimed to solve the scenario where developers would develop code, make significant architectural changes, and not consider the Operations team that needed to deploy the code to production. So when the time came for the operations team to deploy the developers' code changes to production, this would result in a broken deployment, meaning crucial software fixes or new products would not reach customers as planned, and the deployment process would typically take days or weeks to fix.

This led to frustration to all the teams involved, as developers would have to stop coding new features and instead would have to help operations staff fix the deployment process. Operations teams would also be frustrated, as often they would not have been told that infrastructure changes were required to deploy the new release to production. As a result, the operations team did not have the idea to adequately prepare the production environment to support the architectural changes.

This common IT scenario highlights the broken process and operational model that would happen continually and cause friction between development and operations teams.

"DevOps" was an initiative setup to foster an environment of collaboration and communication between these previously conflicting teams. It promotes teams to speak daily, making each other aware of changes and consequently preventing avoidable situations from happening. So, it just so happens that development and operations staff were the first set of silos that DevOps aimed to solve. Consequently, it branded DevOps as a way to unify the teams to work, as one consolidated fluid function, but it could easily have been called something else.

DevOps aims to create better working relationships between teams and a happier working environment, as frankly nobody enjoys conflict or firefighting preventable issues on a daily basis. It also aims to share knowledge between teams to prevent the development teams being viewed as "ignorant to infrastructure" and operations teams to be "blockers to changes" and "slowing down devs". These are the common misconceptions that teams working in silos have of one another when they don't take the time to understand each other's goals.

DevOps strives to build an office environment where teams appreciate other teams and their aims, and they are respectful of their common goals. DevOps is undoubtedly one most talked about topics in the IT industry today. It is not a coincidence that its popularity has risen with the emergence of agile software development, as an alternative to using the more traditional waterfall approach.

Waterfall development and the "V-Model" encompass the separate phases of analysis, design, implementation (coding), and testing. These phases are split up traditionally into different isolated teams, with formalized project hand-off dates that are set in stone.

Agile was born out of the realization that in the fast-paced software industry, long running projects were suboptimal and not the best way of delivering real value to customers. As a result, agile has moved projects to shorter iteration cycles that incorporated analysis, design, implementation, and testing into two-week cycles (sprints) and aimed at using a prototyping approach instead.

The prototyping approach uses the notion of incremental development, which has allowed companies to gather feedback on products earlier in the release cycle, rather than utilizing a big bang approach that delivered the full solution in one big chunk at the end of the implementation phase.

Delivering projects in a waterfall fashion at the end of the waterfall implementation stage ran the risk of delivering products that customers did not want or where developers had misinterpreted requirements. These issues were typically only discovered in the final test phase when the project was ready to be taken to market. This often resulted in projects being deemed a failure or resulting in huge delays, whereas costly rework and change requests could be actioned.

Agile software development for the best part has fostered the need to collapse down those team silos that were typically associated with waterfall software development, and this strengthened the need for daily collaboration.

With the introduction of agile software development, it also has changed the way software testing is carried out too, with the same DevOps principles also being applied to testing functions.

Quality assurance test teams can no longer afford to be reactive either, much like operations teams before them. So, this promoted the need for test teams to work more efficiently and not delay products reaching market. This, however, could not be done at the expense of the product, so they needed to find a way to make sure applications are tested adequately and pass all quality assurance checks while working in a smarter way.

It was readily accepted that quality assurance test teams can no longer operate in silos separate from development teams; instead, agile software development has promoted test cases being written in parallel to the software development, so they are not a separate activity. This is in stark contrast to code being deployed into a test environment and left to a team of testers to execute manual tests or run a set of test packs where they deal with issues reactively.

Agile has promoted developers and quality assurance testers to instead work together in scrum teams on a daily basis to test software before it is packaged for deployment, with those same tests then being maintained and kept up to date and used to seed the regression test packs.

This has been used to mitigate the friction caused by developers checking in code changes that break quality assurance test team's regression packs. With siloed test teams, a common scenario that would often cause friction is be that a graphical user interface (GUI) would suddenly be changed by a developer, resulting in a portion of regression tests breaking. This change would be done without notifying the test team. The tests would be failing because they were written for an old GUI and were suddenly outdated, as opposed to breaking because developers had actually introduced a software failure or a bug.

This reactive approach to testing did not build confidence in the validity of the test failures reported by automated test packs as they are not always conclusively down to a software failure, and this introduced unnecessary delays due to suboptimal IT structure.

Instead if the communication between development and test teams had been better, using the principles promoted by DevOps, then these delays and suboptimal ways of working can be avoided.

More recently, we have seen the emergence of DevSecOps that have looked at integrating security and compliance into the software delivery process, as opposed to being bolted on manual actions and separate reactive initiatives. DevSecOps has looked at using DevOps and agile philosophies and embraced the idea of embedding security engineers in scrum teams to make sure that security requirements are met at the point of inception.

This means that security and compliance can be integrated as automated phases in Continuous Delivery pipelines, to run security compliance checks on every software release, and not slow down the software development lifecycle for developers and generate the necessary feedback loops.

So networking teams can learn from DevOps methodologies too much like development, operations, quality assurance, and security teams. These teams have utilized agile processes to improve their interaction with the software development process and benefit from using feedback loops.

How many times have network engineers had no choice but to delay a software release, as network changes need to be implemented and have been done so inefficiently using ticket-based systems that are not aligned with the processes other departments use? How many times have manually implemented network changes broken production services? This isn't a criticism of network teams or the ability of network engineers; it's the realization that the operational model needs to change and they can.

What this book covers

This book will look at how networking changes can be made more efficient so as not to slow down the software development lifecycle. It will help outline strategies network engineers can adopt to automate network operations. We will focus on setting up network teams to succeed in an automation-driven environment, enabling the teams to work in a more collaborative fashion, and improve efficiency.

It will also show that network teams need to build new skills and learn configuration management tools such as Ansible to help them achieve this goal. The book will show the advantages that these tools bring, using the network modules they provide, and that they will help make automation easy and act as a self-starter guide.

We will focus on some of the cultural challenges that need to be overcome to influence and implement automation processes for network functions and convince network teams to make the most of networking APIs that are now provided by vendors can be trusted.

The book will discuss public and private clouds such as AWS and OpenStack, and ways they are used to provide networking to users. It will also discuss the emergence of software-defined networking solutions, such as Juniper Contrail, VMWare NSX, CISCO ACI, and focus on the Nokia Nuage VSP solution, which aims to make networking functions a self-service commodity.

The book will also highlight how continuous integration and delivery processes and deployment pipelines can be applied to govern network changes. It will also show ways that unit testing can be applied to automated network changes to integrate them with the software delivery lifecycle.

A detailed chapter overview for the book is detailed below:

Chapter 1, The Impact of Cloud on Networking, will discuss ways in which the emergence of AWS for public cloud and OpenStack for private cloud have changed the way developers want to consume networking. It will look at some of the networking services AWS and OpenStack provide out of the box and look at some of the networking features they provide. It will show examples of how these cloud platforms have made networking a commodity much like infrastructure.

Chapter 2, The Emergence of Software-defined Networking, will discuss how software-defined networking has emerged. It will look at the methodology and focus on some of the scaling benefits and features this provides over and above the out-of-the-box experience from AWS and OpenStack. It will illustrate how one of the market-leading SDN solutions, Nuage, applies these concepts and principles and discusses other SDN solutions on the market.

Chapter 3, Bringing DevOps to Network Operations, will detail the pros and cons of a top-down and bottom-up DevOps initiatives with regards to networking. It will give readers food for thought on some of the strategies that have been a success and which ones have typically failed. This chapter will help CTOs, senior managers, and engineers who are trying to initiate a DevOps model in their company's network department and outline some of the different strategies they could use to achieve the cultural changes they desire.

Chapter 4, Configuring Network Devices Using Ansible, will outline the benefits of using configuration management tools to install and push configuration to network devices and discuss some of the open source network modules available to do this at the moment and how they work. It will give some examples of process flows that could be adopted to maintain device configuration.

Chapter 5, Orchestrating Load Balancers Using Ansible, will describe the benefits of using Ansible to orchestrate load balancers and the approaches to roll new software releases into service without the need for downtime or manual intervention. It will give some examples of some process flows that could be adopted to allow orchestration of both immutable and static servers looking at the different load balancer technologies available.

Chapter 6, Orchestrating SDN Controllers Using Ansible, will outline the benefits of using Ansible to orchestrate SDN controllers. It will outline the benefits of software-defined networking and why it is paramount to automate the network functions that an SDN controller exposes. This includes setting ACL rules dynamically, which will allow network engineers to provide a Network as a Service (NaaS) allowing developers to self-service their networking needs. It will discuss deployment strategies such as blue green networks as well as exploring some of the process flows that could be used to implement a NaaS approach.

Chapter 7, Using Continuous Integration Builds for Network Configuration, will discuss moving to a model where network configuration is stored in source control management systems, so it is easily audited and versioned and changes can be rolled back.

It will look at workflows that can be used to set up network configuration CI builds using tools such as Jenkins and Git.

Chapter 8, Testing Network Changes, will outline the importance of using test environments to test network changes before applying them in production. It will explore some of the open source tooling available and walk through some of the test strategies that can be applied to make sure that network changes are thoroughly tested before applying them to production.

Chapter 9, Using Continuous Delivery Pipelines to Deploy Network Changes, will show readers how to use continuous integration and Continuous Delivery pipelines to deliver network changes to production and put them through associated test environments. It will give some examples of some process flows that could be adopted to deliver network changes to production and how they can easily sit alongside infrastructure and code changes in deployment pipelines.

Chapter 10, The Impact of Containers on Networking, dedicated container technologies such as Docker and container orchestration engines such as Kubernetes and Swarm are becoming more and more popular with companies that are moving to microservice architectures. As a result, this has changed networking requirements. This chapter will look at how containers operate and the impact they have had on networking.

Chapter 11, Securing the Network, will look at how this approach makes a security engineer's job of auditing the network easier. It will look at the possible attack vectors in a software-defined network and ways that security checks can be integrated into a DevOps model.

What you need for this book

This book assumes a medium level on networking knowledge, a basic level of Linux knowledge, a basic knowledge of cloud computing technologies, and a broad knowledge of IT. It is focusing primarily on particular process workflows that can be implemented rather than base technologies, so the ideas and content can be applied to any organization, no matter the technology that is used.

However, that being said, it could be beneficial to readers to access the following technologies when digesting some of the chapters' content:

- AWS <https://aws.amazon.com/free/>
- OpenStack <http://trystack.org/>
- Nuage VSP <http://nuagex.io/>

Who this book is for

The target audience for this book is network engineers who want to automate the manual and repetitive parts of their job or developers or system admins who want to automate all network functions.

This book will also provide a good insight to CTOs or managers who want to understand ways in which they can make their network departments more agile and initiate real cultural change within their organizations.

The book will also aid casual readers who want to understand more about DevOps, continuous integration, and Continuous Delivery and how they can be applied to real-world scenarios as well as insights on some of the tooling that is available to facilitate automation.

Conventions



In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.



Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "These services are then bound to the `lbvserver` entity."

Any command-line input or output is written as follows:

```
ansible-playbook -I ineventories/openstack.py -l qa -e environment=qa
-e current_build=9 playbooks/add_hosts_to_netscaler.yml
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Click the **Search** button on Google."

 Warnings or important notes appear in a box like this. 

 Tips and tricks appear like this. 

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from http://www.packtpub.com/sites/default/files/downloads/DevOpsforNetworking_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

The Impact of Cloud on Networking

This chapter will look at ways that networking has changed in the private data centers and evolved in the past few years. It will focus on the emergence of **Amazon Web Services (AWS)** for public cloud and **OpenStack** for private cloud and ways in which this has changed the way developers want to consume networking. It will look at some of the networking services that AWS and OpenStack provide out of the box and look at some of the features they provide. It will show examples of how these cloud platforms have made networking a commodity much like infrastructure.

In this chapter, the following topics will be covered:

- An overview of cloud approaches
- The difference between Spanning Tree networks and Leaf-Spine networking
- Changes that have occurred in networking with the introduction of public cloud
- The Amazon Web Services approach to networking
- The OpenStack approach to networking

An overview of cloud approaches

The cloud provider market is currently saturated with a multitude of different private, public, and hybrid cloud solutions, so choice is not a problem for companies looking to implement public, private, or hybrid cloud solutions.

Consequently, choosing a cloud solution can sometimes be quite a daunting task, given the array of different options that are available.

The battle between public and private cloud is still in its infancy, with only around 25 percent of the industry using public cloud, despite its perceived popularity, with solutions such as Amazon Web Services, Microsoft Azure, and Google Cloud taking a large majority of that market share. However, this still means that 75 percent of the cloud market share is available to be captured, so the cloud computing market will likely go through many iterations in the coming years.

So why are many companies considering public cloud in the first place and why does it differ from private and hybrid clouds?

Public clouds

Public clouds are essentially a set of data centers and infrastructure that are made publicly available over the Internet to consumers. Despite its name, it is not magical or fluffy in any way. Amazon Web Services launched their public cloud based on the idea that they could rent out their servers to other companies when they were not using them during busy periods of the year.

Public cloud resources can be accessed via a **Graphical User Interface (GUI)** or, programmatically, via a set of API endpoints. This allows end users of the public cloud to create infrastructure and networking to host their applications.

Public clouds are used by businesses for various reasons, such as the speed it takes to configure and using public cloud resources is relatively low. Once credit card details have been provided on a public cloud portal, end users have the freedom to create their own infrastructure and networking, which they can run their applications on.

This infrastructure can be elastically scaled up and down as required, all at a cost of course to the credit card.

Public cloud has become very popular as it removes a set of historical impediments associated with **shadow IT**. Developers are no longer hampered by the restrictions enforced upon them by bureaucratic and slow internal IT processes. Therefore, many businesses are seeing public cloud as a way to skip over these impediments and work in a more agile fashion allowing them to deliver new products to market at a greater frequency.

When a business moves its operations to a public cloud, they are taking the bold step to stop hosting their own data centers and instead use a publicly available public cloud provider, such as Amazon Web Services, Microsoft Azure, IBM BlueMix, Rackspace, or Google Cloud.

The reliance is then put upon the public cloud for uptime and **Service Level Agreements (SLA)**, which can be a huge cultural shift for an established business.

Businesses that have moved to public cloud may find they no longer have a need for a large internal infrastructure team or network team, instead all infrastructure and networking is provided by the third-party public cloud, so it can in some quarters be viewed as giving up on internal IT.

Public cloud has proved a very successful model for many start-ups, given the agility it provides, where start-ups can put out products quickly using software-defined constructs without having to set up their own data center and remain product focused.

However, the **Total Cost of Ownership (TCO)** to run all of a business's infrastructure in a public cloud is a hotly debated topic, which can be an expensive model if it isn't managed and maintained correctly. The debate over public versus private cloud TCO rages on as some argue that public cloud is a great short-term fix but growing costs over a long period of time mean that it may not be a viable long-term solution compared with private cloud.

Private cloud

Private cloud is really just an extension of the initial benefits introduced by virtualization solutions, such as VMware, Hyper-V, and Citrix Xen, which were the cornerstone of the virtualization market. The private cloud world has moved on from just providing virtual machines, to providing software-defined networking and storage.

With the launch of public clouds, such as Amazon Web Services, private cloud solutions have sought to provide like-for-like capability by putting a software-defined layer on top of their current infrastructure. This infrastructure can be controlled in the same way as the public cloud via a GUI or programmatically using APIs.

Private cloud solutions such as Apache CloudStack and open source solutions such as OpenStack have been created to bridge the gap between the private cloud and the public cloud.

This has allowed vendors the agility of private cloud operations in their own data center by overlaying software-defined constructs on top of their existing hardware and networks.

However, the major benefit of private cloud is that this can be done within the security of a company's own data centers. Not all businesses can use public cloud for compliance, regularity, or performance reasons, so private cloud is still required for some businesses for particular workloads.

Hybrid cloud

Hybrid cloud can often be seen as an amalgamation of multiple clouds. This allows a business to seamlessly run workloads across multiple clouds linked together by a network fabric. The business could select the placement of workloads based on cost or performance metrics.

A hybrid cloud can often be made up of private and public clouds. So, as an example, a business may have a set of web applications that it wishes to scale up for particular busy periods and are better suited to run on public cloud so they are placed there. However, the business also needs a highly regulated, PCI-compliant database, which would be better-suited to being deployed in a private on-premises cloud. So a true hybrid cloud gives a business these kinds of options and flexibility.

Hybrid cloud really works on the premise of using different clouds for different use cases, where each horse (application workload) needs to run a particular course (cloud). So, sometimes, a vendor-provided **Platform as a Service (PaaS)** layer can be used to place workloads across multiple clouds or alternately different configuration management tools, or container orchestration technologies can be used to orchestrate application workload placement across clouds.

Software-defined

The choice between public, private, or hybrid cloud really depends on the business, so there is no real right or wrong answer. Companies will likely use hybrid cloud models as their culture and processes evolve over the next few years.

If a business is using a public, private, or hybrid cloud, the common theme with all implementations is that they are moving towards a software-defined operational model.

So what does the term *software-defined* really mean? In simple terms, *software-defined* means running a software abstraction layer over hardware. This software abstraction layer allows graphical or programmatic control of the hardware. So, constructs, such as infrastructure, storage, and networking, can be software defined to help simplify operations, manageability as infrastructure and networks scale out.

When running private clouds, modifications need to be made to incumbent data centers to make them private cloud ready; sometimes, this is important, so the private data center needs to evolve to meet those needs.

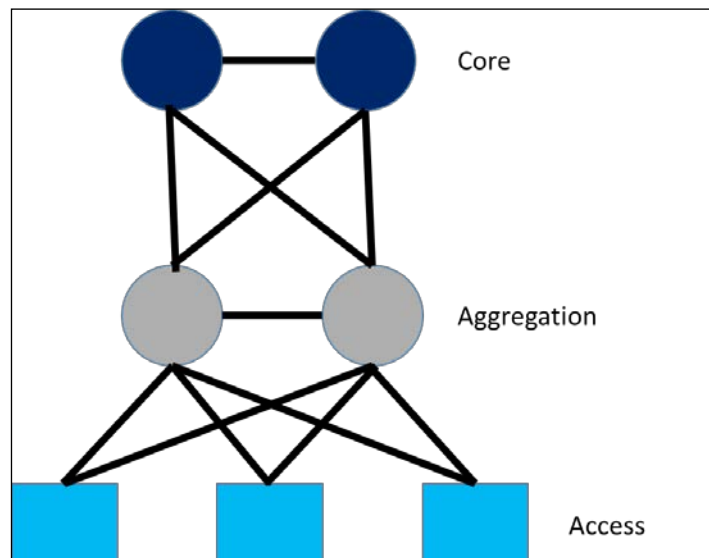
The difference between Spanning Tree and Leaf-Spine networking

When considering the private cloud, traditionally, company's private datacenters have implemented 3-tier layer 2 networks based on the **Spanning Tree Protocol (STP)**, which doesn't lend itself well to modern software-defined networks. So, we will look at what a STP is in more depth as well as modern Leaf-Spine network architectures.

Spanning Tree Protocol

The implementation of STP provides a number of options for network architects in terms of implementation, but it also adds a layer of complexity to the network. Implementation of the STP gives network architects the certainty that it will prevent layer 2 loops from occurring in the network.

A typical representation of a 3-tier layer 2 STP-based network can be shown as follows:



- The **Core** layer provides routing services to other parts of the data center and contains the core switches
- The **Aggregation** layer provides connectivity to adjacent **Access** layer switches and the top of the Spanning Tree core

The bottom of the tree is the **Access** layer; this is where bare metal (physical) or virtual machines connect to the network and are segmented using different VLANs.

The use of layer 2 networking and STP mean that at the access layer of the network will use VLANs spread throughout the network. The VLANs sit at the access layer, which is where virtual machines or bare metal servers are connected. Typically, these VLANs are grouped by type of application, and firewalls are used to further isolate and secure them.

Traditional networks are normally segregated into some combination of the following:

- **Frontend:** It typically has web servers that require external access
- **Business Logic:** This often contains stateful services
- **Backend:** This typically contains database servers

Applications communicate with each other by tunneling between these firewalls, with specific **Access Control List (ACL)** rules that are serviced by network teams and governed by security teams.

When using STP in a layer 2 network, all switches go through an election process to determine the root switch, which is granted to the switch with the lowest bridge ID, with a bridge ID encompassing the bridge priority and MAC address of the switch.

Once elected, the root switch becomes the base of the spanning tree; all other switches in the Spanning Tree are deemed non-root will calculate their shortest path to the root and then block any redundant links, so there is one clear path. The calculation process to work out the shortest path is referred to as network convergence. (For more information refer to the following link: <http://etutorials.org/Networking/Lan+switching+fundamentals/Chapter+10.+Implementing+and+Tuning+Spanning+Tree/Spanning-Tree+Convergence/>)

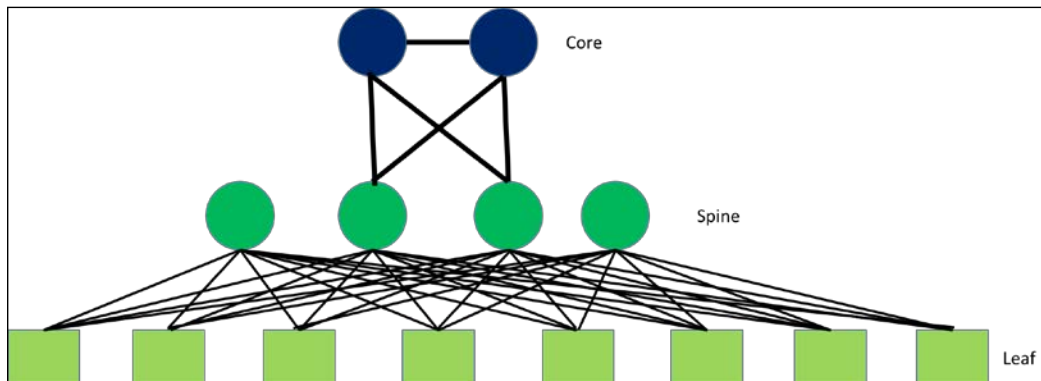
Network architects designing the layer 2 Spanning Tree network need to be careful about the placement of the root switch, as all network traffic will need to flow through it, so it should be selected with care and given an appropriate bridge priority as part of the network reference architecture design. If at any point, switches have been given the same bridge priority then the bridge with the lowest MAC address wins.

Network architects should also design the network for redundancy so that if a root switch fails, there is a nominated backup root switch with a priority of one value less than the nominated root switch, which will take over when a root switch fails. In the scenario, the root switch fails the election process will begin again and the network will converge, which can take some time.

The use of STP is not without its risks, if it does fail due to user configuration error, data center equipment failure or software failure on a switch or bad design, then the consequences to a network can be huge. The result can be that loops might form within the bridged network, which can result in a flood of broadcast, multicast or unknown-unicast storms that can potentially take down the entire network leading to long network outages. The complexity associated with network architects or engineers troubleshooting STP issues is important, so it is paramount that the network design is sound.

Leaf-Spine architecture

In recent years with the emergence of cloud computing, we have seen data centers move away from a STP in favor of a Leaf-Spine networking architecture. The Leaf-Spine architecture is shown in the following diagram:



In a Leaf-Spine architecture:

- Spine switches are connected into a set of core switches
- Spine switches are then connected with Leaf switches with each Leaf switch deployed at the top of rack, which means that any Leaf switch can connect to any Spine switch in one hop

Leaf-Spine architectures are promoted by companies such as Arista, Juniper, and Cisco. A Leaf-Spine architecture is built on layer 3 routing principle to optimize throughput and reduce latency.

Both Leaf and Spine switches communicate with each other via **external Border Gate Protocol (eBGP)** as the routing protocol for the IP fabric. eBGP establishes a **Transmission Control Protocol (TCP)** connection to each of its BGP peers before BGP updates can be exchanged between the switches. Leaf switches in the implementation will sit at top of rack and can be configured in **Multichassis Link Aggregation (MLAG)** mode using **Network Interface Controller (NIC)** bonding.

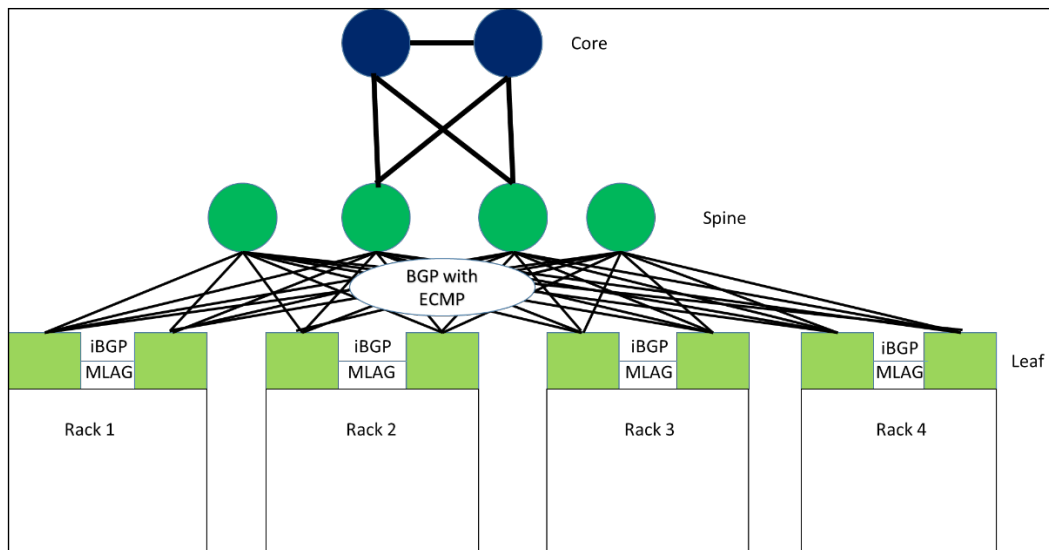
MLAG was originally used with **STP** so that two or more switches are bonded to emulate like a single switch and used for redundancy so they appeared as one switch to STP. In the event of a failure this provided multiple uplinks for redundancy in the event of a failure as the switches are peered, and it worked around the need to disable redundant paths. Leaf switches can often have **internal Border Gate Protocol (iBGP)** configured between the pairs of switches for resiliency.

In a Leaf-Spine architecture, Spine switches do not connect to other Spine switches, and Leaf switches do not connect directly to other Leaf switches unless bonded top of rack using MLAG NIC bonding. All links in a Leaf-Spine architecture are set up to forward with no looping. Leaf-Spine architectures are typically configured to implement **Equal Cost Multipathing (ECMP)**, which allows all routes to be configured on the switches so that they can access any Spine switch in the layer 3 routing fabric.

ECMP means that Leaf switches routing table has the next-hop configured to forward to each Spine switch. In an ECMP setup, each leaf node has multiple paths of equal distance to each Spine switch, so if a Spine or Leaf switch fails, there is no impact as long as there are other active paths to another adjacent Spine switches. ECMP is used to load balance flows and supports the routing of traffic across multiple paths. This is in contrast to the STP, which switches off all but one path to the root when the network converges.

Normally, Leaf-Spine architectures designed for high performance use 10G access ports at Leaf switches mapping to 40G Spine ports. When device port capacity becomes an issue, new Leaf switches can be added by connecting it to every Spine on the network while pushing the new configuration to every switch. This means that network teams can easily scale out the network horizontally without managing or disrupting the switching protocols or impacting the network performance.

An illustration of the protocols used in a Leaf-Spine architecture are shown later, with Spine switches connected to Leaf switches using BGP and ECMP and Leaf switches sitting top of rack and configured for redundancy using MLAG and iBGP:



The benefits of a Leaf-Spine architecture are as follows:

- Consistent latency and throughput in the network
- Consistent performance for all racks
- Network once configured becomes less complex
- Simple scaling of new racks by adding new Leaf switches at top of rack
- Consistent performance, subscription, and latency between all racks
- East-west traffic performance is optimized (virtual machine to virtual machine communication) to support microservice applications
- Removes VLAN scaling issues, controls broadcast and fault domains

The one drawback of a Leaf-Spine topology is the amount of cables it consumes in the data center.

OVSDB

Modern switches have now moved towards open source standards, so they can use the same pluggable framework. The open standard for virtual switches is **Open vSwitch**, which was born out of the necessity to come up with an open standard that allowed a virtual switch to forward traffic to different virtual machines on the same physical host and physical network. Open vSwitch uses **Open vSwitch database (OVSDB)** that has a standard extensible schema.

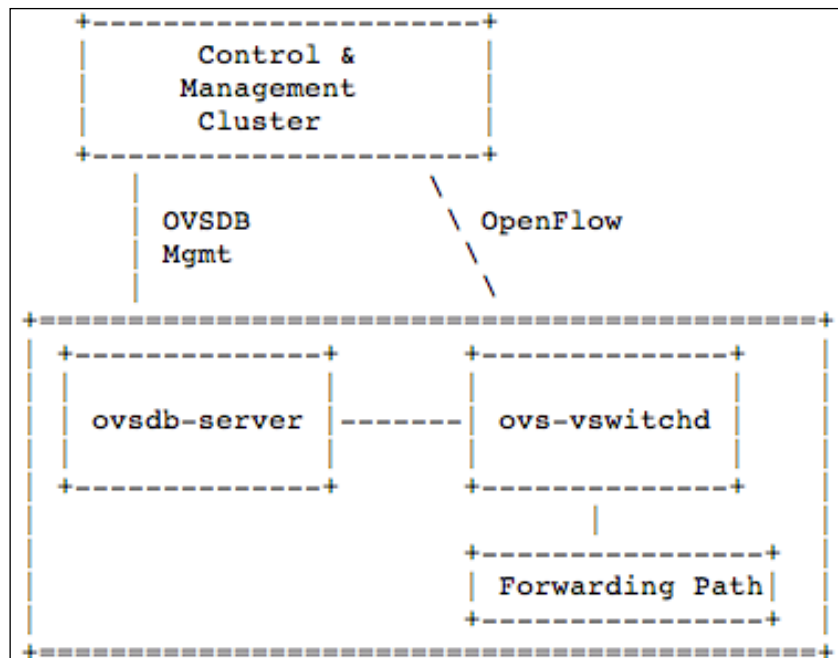
Open vSwitch was initially deployed at the hypervisor level but is now being used in container technology too, which has Open vSwitch implementations for networking.

The following hypervisors currently implement Open vSwitch as their virtual switching technology:

- KVM
- Xen
- Hyper-V

Hyper-V has recently moved to support Open vSwitch using the implementation created by Cloudbase (<https://cloudbase.it/>), which is doing some fantastic work in the open source space and is testament to how Microsoft's business model has evolved and embraced open source technologies and standards in recent years. Who would have thought it? Microsoft technologies now run natively on Linux.

The Open vSwitch exchanges **OpenFlow** between virtual switch and physical switches in order to communicate and can be programmatically extended to fit the needs of vendors. In the following diagram, you can see the Open vSwitch architecture. Open vSwitch can run on a server using the KVM, Xen, or Hyper-V virtualization layer:



The **ovsdb-server** contains the OVSDB schema that holds all switching information for the virtual switch. The **ovs-vsitchd** daemon talks **OpenFlow** to any **Control & Management Cluster**, which could be any SDN controller that can communicate using the **OpenFlow** protocol.

Controllers use OpenFlow to install flow state on the virtual switch, and OpenFlow dictates what actions to take when packets are received by the virtual switch.

When Open vSwitch receives a packet it has never seen before and has no matching flow entries, it sends this packet to the controller. The controller then makes a decision on how to handle this packet based on the flow rules to either block or forward. The ability to configure **Quality of Service (QoS)** and other statistics is possible on Open vSwitch.

Open vSwitch is used to configure security rules and provision ACL rules at the switch level on a hypervisor.

A Leaf-Spine architecture allows overlay networks to be easily built, meaning that cloud and tenant environments are easily connected to the layer 3 routing fabric. Hardware **Vxlan Tunnel Endpoints (VTEPs)** IPs are associated with each Leaf switch or a pair of Leaf switches in MLAG mode and are connected to each physical compute host via **Virtual Extensible LAN (VXLAN)** to each Open vSwitch that is installed on a hypervisor.

This allows an SDN controller, which is provided by vendors, such as Cisco, Nokia, and Juniper to build an overlay network that creates VXLAN tunnels to the physical hypervisors using Open vSwitch. New VXLAN tunnels are created automatically if a new compute is scaled out, then SDN controllers can create new VXLAN tunnels on the Leaf switch as they are peered with the Leaf switch's hardware **VXLAN Tunnel End Point (VTEP)**.

Modern switch vendors, such as Arista, Cisco, Cumulus, and many others, use OVSDB, and this allows SDN controllers to integrate at the **Control & Management Cluster** level. As long as an SDN controller uses OVSDB and OpenFlow protocol, they can seamlessly integrate with the switches and are not tied into specific vendors. This gives end users a greater depth of choice when choosing switch vendors and SDN controllers, which can be matched up as they communicate using the same open standard protocol.

Changes that have occurred in networking with the introduction of public cloud

It is unquestionable that the emergence of the AWS, which was launched in 2006, changed and shaped the networking landscape forever. AWS has allowed companies to rapidly develop their products on the AWS platform. AWS has created an innovative set of services for end users, so they can manage infrastructure, load balancing, and even databases. These services have led the way in making the DevOps ideology a reality, by allowing users to elastically scale up and down infrastructure. They need to develop products on demand, so infrastructure wait times are no longer an inhibitor to development teams. AWS rich feature set of technology allows users to create infrastructure by clicking on a portal or more advanced users that want to programmatically create infrastructure using configuration management tooling, such as **Ansible, Chef, Puppet, Salt** or **Platform as a Service (PaaS)** solutions.

An overview of AWS

In 2016, the AWS **Virtual Private Cloud (VPC)** secures a set of Amazon EC2 instances (virtual machines) that can be connected to any existing network using a VPN connection. This simple construct has changed the way that developers want and expect to consume networking.

In 2016, we live in a consumer-based society with mobile phones allowing us instant access to the Internet, films, games, or an array of different applications to meet our every need, instant gratification if you will, so it is easy to see the appeal of AWS has to end users.

AWS allows developers to provision instances (virtual machines) in their own personal network, to their desired specification by selecting different flavors (CPU, RAM, and disk) using a few button clicks on the AWS portal's graphical user interface, alternately using a simple call to an API or scripting against the AWS-provided SDKs.

So now a valid question, why should developers be expected to wait long periods of time for either infrastructure or networking tickets to be serviced in on-premises data centers when AWS is available? It really shouldn't be a hard question to answer. The solution surely has to either be moved to AWS or create a private cloud solution that enables the same agility. However, the answer isn't always that straightforward, there are following arguments against using AWS and public cloud:

- Not knowing where the data is actually stored and in which data center
- Not being able to hold sensitive data offsite
- Not being able to assure the necessary performance
- High running costs

All of these points are genuine blockers for some businesses that may be highly regulated or need to be PCI compliant or are required to meet specific regularity standards. These points may inhibit some businesses from using public cloud so as with most solutions it isn't the case of one size fits all.

In private data centers, there is a cultural issue that teams have been set up to work in silos and are not set up to succeed in an agile business model, so a lot of the time using AWS, Microsoft Azure, or Google Cloud is a quick fix for broken operational models.

Ticketing systems, a staple of broken internal operational models, are not a concept that aligns itself to speed. An IT ticket raised to an adjacent team can take days or weeks to complete, so requests are queued before virtual or physical servers can be provided to developers. Also, this is prominent for network changes too, with changes such as a simple modification to ACL rules taking an age to be implemented due to ticketing backlogs.

Developers need to have the ability to scale up servers or prototype new features at will, so long wait times for IT tickets to be processed hinder delivery of new products to market or bug fixes to existing products. It has become common in internal IT that some **Information Technology Infrastructure Library (ITIL)** practitioners put a sense of value on how many tickets that processed over a week as the main metric for success. This shows complete disregard for customer experience of their developers. There are some operations that need to shift to the developers, which have traditionally lived with internal or shadow IT, but there needs to be a change in operational processes at a business level to invoke these changes.

Put simply, AWS has changed the expectations of developers and the expectations placed on infrastructure and networking teams. Developers should be able to service their needs as quickly as making an alteration to an application on their mobile phone, free from slow internal IT operational models associated with companies.

But for start-ups and businesses that can use AWS, which aren't constrained by regulatory requirements, it skips the need to hire teams to rack servers, configure network devices, and pay for the running costs of data centers. It means they can start viable businesses and run them on AWS by putting in credit card details the same way as you would purchase a new book on Amazon or eBay.

OpenStack overview

The reaction to AWS was met with trepidation from competitors, as it disrupted the cloud computing industry and has led to PaaS solutions such as **Cloud Foundry** and **Pivotal** coming to fruition to provide an abstraction layer on top of hybrid clouds.

When a market is disrupted, it promotes a reaction, from it spawned the idea for a new private cloud. In 2010, a joint venture by Rackspace and NASA, launched an open source cloud-software initiative known as OpenStack, which came about as NASA couldn't put their data in a public cloud.

The OpenStack project intended to help organizations offer cloud computing services running on standard hardware and directly set out to mimic the model provided by AWS. The main difference with OpenStack is that it is an open source project that can be used by leading vendors to bring AWS-like ability and agility to the private cloud.

Since its inception in 2010, OpenStack has grown to have over 500 member companies as part of the OpenStack Foundation, with platinum members and gold members that comprise the biggest IT vendors in the world that are actively driving the community.

The platinum members of the OpenStack foundation are:



OpenStack is an open source project, which means its source code is publicly available and its underlying architecture is available for analysis, unlike AWS, which acts like a magic box of tricks but it is not really known for how it works underneath its shiny exterior.

OpenStack is primarily used to provide an **Infrastructure as a Service (IaaS)** function within the private cloud, where it makes commodity x86 compute, centralized storage, and networking features available to end users to self-service their needs, be it via the horizon dashboard or through a set of common API's.

Many companies are now implementing OpenStack to build their own data centers. Rather than doing it on their own, some companies are using different vendor hardened distributions of the community upstream project. It has been proven that using a vendor hardened distributions of OpenStack, when starting out, mean that OpenStack implementation is far likelier to be successful. Initially, for some companies, implementing OpenStack can be seen as complex as it is a completely new set of technology that a company may not be familiar with yet. OpenStack implementations are less likely to fail when using professional service support from known vendors, and it can create a viable alternative to enterprise solutions, such as AWS or Microsoft Azure.

Vendors, such as Red Hat, HP, Suse, Canonical, Mirantis, and many more, provide different distributions of OpenStack to customers, complete with different methods of installing the platform. Although the source code and features are the same, the business model for these OpenStack vendors is that they harden OpenStack for enterprise use and their differentiator to customers is their professional services.

There are many different OpenStack distributions available to customers with the following vendors providing OpenStack distributions:

- Bright Computing
- Canonical
- HPE
- IBM
- Mirantis
- Oracle OpenStack for Oracle Linux, or O3L
- Oracle OpenStack for Oracle Solaris
- Red Hat
- SUSE
- VMware Integrated OpenStack (VIO)

OpenStack vendors will support build out, on-going maintenance, upgrades, or any customizations a client needs, all of which are fed back to the community. The beauty of OpenStack being an open source project is that if vendors customize OpenStack for clients and create a real differentiator or competitive advantage, they cannot fork OpenStack or uniquely sell this feature. Instead, they have to contribute the source code back to the upstream open source OpenStack project.

This means that all competing vendors contribute to its success of OpenStack and benefit from each other's innovative work. The OpenStack project is not just for vendors though, and everyone can contribute code and features to push the project forward.

OpenStack maintains a release cycle where an upstream release is created every six months and is governed by the OpenStack Foundation. It is important to note that many public clouds, such as AT&T, RackSpace, and GoDaddy, are based on OpenStack too, so it is not exclusive to private clouds, but it has undeniably become increasingly popular as a private cloud alternative to AWS public cloud and now widely used for **Network Function Virtualization (NFV)**.

So how does AWS and OpenStack work in terms of networking? Both AWS and OpenStack are made up of some mandatory and optional projects that are all integrated to make up its reference architecture. Mandatory projects include compute and networking, which are the staple of any cloud solution, whereas others are optional bolt-ons to enhance or extend capability. This means that end users can cherry-pick the projects they are interested in to make up their own personal portfolio.

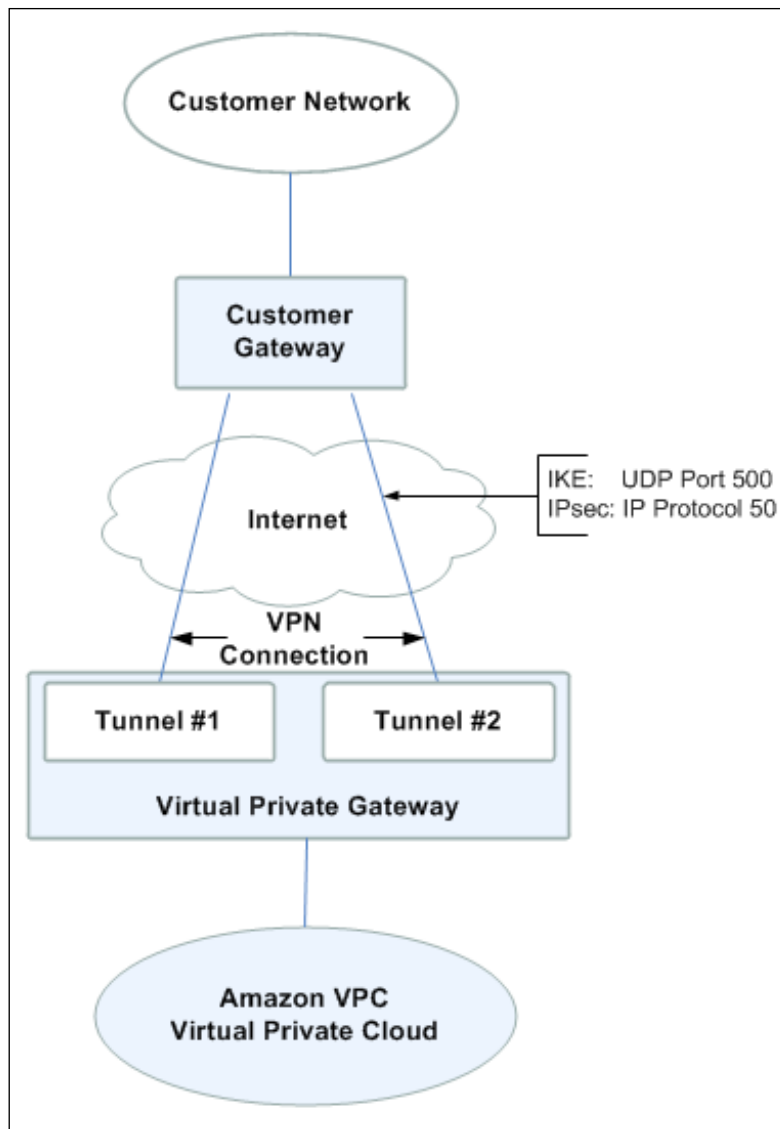
The AWS approach to networking

Having discussed both AWS and OpenStack, first, we will explore the AWS approach to networking, before looking at an alternative method using OpenStack and compare the two approaches. When first setting up networking in AWS, a tenant network in AWS is instantiated using VPC, which post 2013 deprecated AWS classic mode; but what is VPC?

Amazon VPC

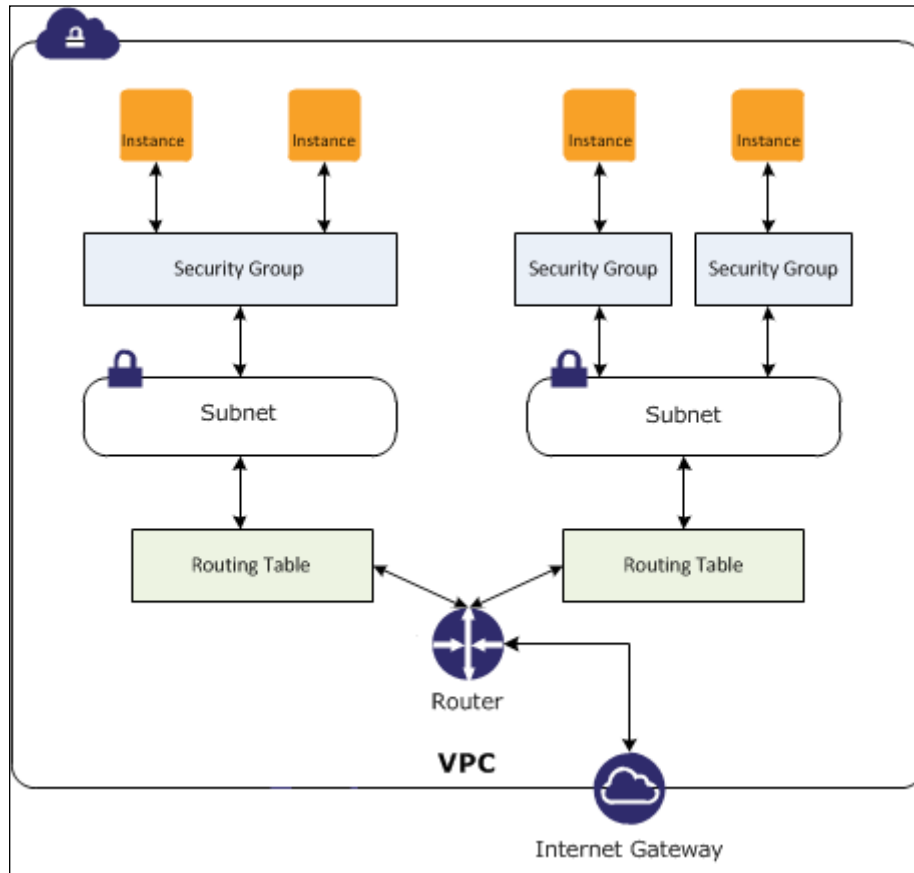
A VPC is the new default setting for new customers wishing to access AWS. VPCs can also be connected to customer networks (private data centers) by allowing AWS cloud to extend a private data center for agility. The concept of connecting a private data center to an AWS VPC is using something AWS refers to as a customer gateway and virtual private gateway. A virtual private gateway in simple terms is just two redundant VPN tunnels, which are instantiated from the customer's private network.

Customer gateways expose a set of external static addresses from a customer site, which are typically **Network Address Translation-Traversal (NAT-T)** to hide the source address. UDP port 4500 should be accessible in the external firewall in the private data center. Multiple VPCs can be supported from one customer gateway device.



A VPC gives an isolated view of everything an AWS customer has provisioned in AWS public cloud. Different user accounts can then be set up against VPC using the AWS **Identity and Access Management (IAM)** service, which has customizable permissions.

The following example of a VPC shows instances (virtual machines) mapped with one or more security groups and connected to different subnets connected to the VPC router:



A VPC simplifies networking greatly by putting the constructs into software and allows users to perform the following network functions:

- Creating instances (virtual machines) mapped to subnets
- Creating **Domain Name System (DNS)** entries that are applied to instances
- Assigning public and private IP addresses
- Creating or associating subnets
- Creating custom routing
- Applying security groups with associated ACL rules

By default, when an instance (virtual machine) is instantiated in a VPC, it will either be placed on a default subnet or custom subnet if specified.

All VPCs come with a default router when the VPC is created, the router can have additional custom routes added and routing priority can also be set to forward traffic to particular subnets.

Amazon IP addressing

When an instance is spun up in AWS, it will automatically be assigned a mandatory private IP address by **Dynamic Host Configuration Protocol (DHCP)** as well as a public IP and DNS entry too unless dictated otherwise. Private IPs are used in AWS to route east-west traffic between instances when virtual machine needs to communicate with adjacent virtual machines on the same subnet, whereas public IPs are available through the Internet.

If a persistent public IP address is required for an instance, AWS offers the elastic IP addresses feature, which is limited to five per VPC account, which any failed instances IP address can be quickly mapped to another instance. It is important to note that it can take up to 24 hours for a public IP address's **DNS Time To Live (TTL)** to propagate when using AWS.

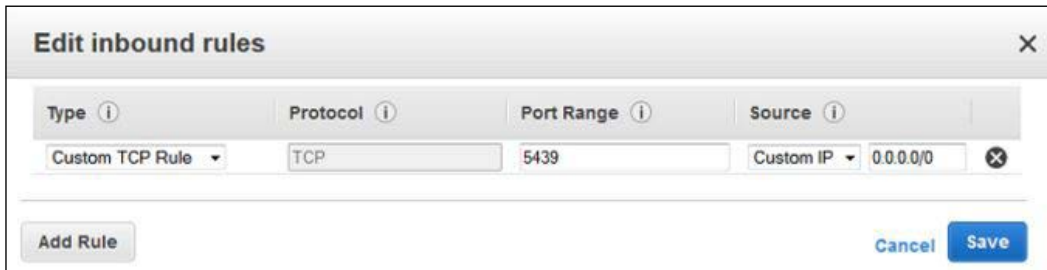
In terms of throughput, AWS instances can support a **Maximum Transmission Unit (MTU)** of 1,500 that can be passed to an instance in AWS, so this needs to be considered when considering application performance.

Amazon security groups

Security groups in AWS are a way of grouping permissive ACL rules, so don't allow explicit denies. AWS security groups act as a virtual firewall for instances, and they can be associated with one or more instances' network interfaces. In a VPC, you can associate a network interface with up to five security groups, adding up to 50 rules to a security group, with a maximum of 500 security groups per VPC. A VPC in an AWS account automatically has a default security group, which will be automatically applied if no other security groups are specified.

Default security groups allow all outbound traffic and all inbound traffic only from other instances in a VPC that also use the default security group. The default security group cannot be deleted. Custom security groups when first created allow no inbound traffic, but all outbound traffic is allowed.

Permissive ACL rules associated with security groups govern inbound traffic and are added using the AWS console (GUI) as shown later in the text, or they can be programmatically added using APIs. Inbound ACL rules associated with security groups can be added by specifying type, protocol, port range, and the source address. Refer to the following screenshot:



Amazon regions and availability zones

A VPC has access to different regions and availability zones of shared compute, which dictate the data center that the AWS instances (virtual machines) will be deployed in. Regions in AWS are geographic areas that are completely isolated by design, where availability zones are isolated locations in that specific region, so an availability zone is a subset of a region.

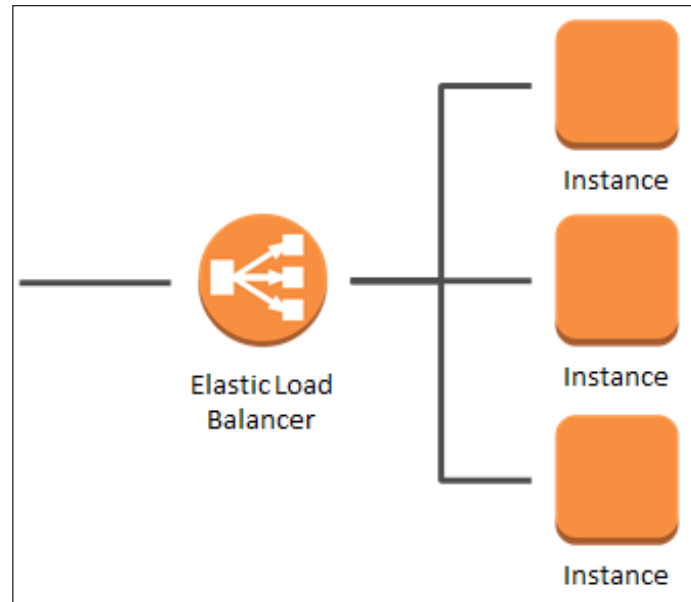
AWS gives users the ability to place their resources in different locations for redundancy as sometimes the health of a specific region or availability zone can suffer issues. Therefore, AWS users are encouraged to use more than one availability zones when deploying production workloads on AWS. Users can choose to replicate their instances and data across regions if they choose to.

Within each isolated AWS region, there are child availability zones. Each availability zone is connected to sibling availability zones using low latency links. All communication from one region to another is across the public Internet, so using geographically distant regions will acquire latency and delay. Encryption of data should also be considered when hosting applications that send data across regions.

Amazon Elastic Load Balancing

AWS also allows **Elastic Load Balancing (ELB)** to be configured within a VPC as a bolt-on service. ELB can either be internal or external. When ELB is external, it allows the creation of an Internet-facing entry point into your VPC using an associated DNS entry and balances load between different instances. Security groups are assigned to ELBs to control the access ports that need to be used.

The following image shows an elastic load balancer, load balancing 3 instances:



The OpenStack approach to networking

Having considered AWS networking, we will now explore OpenStack's approach to networking and look at how its services are configured.

OpenStack is deployed in a data center on multiple controllers. These controllers contain all the OpenStack services, and they can be installed on either virtual machines, bare metal (physical) servers, or containers. The OpenStack controllers should host all the OpenStack services in a highly available and redundant fashion when they are deployed in production.

Different OpenStack vendors provide different installers to install OpenStack. Some examples of installers from the most prominent OpenStack distributions are RedHat Director (based on OpenStack TripleO), Mirantis Fuel, HP's HPE installer (based on Ansible), and Juju for Canonical, which all install OpenStack controllers and are used to scale out compute nodes on the OpenStack cloud acting as an OpenStack workflow management tool.

OpenStack services

A breakdown of the core OpenStack services that are installed on an OpenStack controller are as follows:

- **Keystone** is the identity service for OpenStack that allows user access, which issues tokens, and can be integrated with LDAP or Active directory.
- **Heat** is the orchestration provisioning tool for OpenStack infrastructure.
- **Glance** is the image service for OpenStack that stores all image templates for virtual machines or bare metal servers.
- **Cinder** is the block storage service for OpenStack that allows centralized storage volumes to be provisioned and attached to vms or bare metal servers that can then be mounted.
- **Nova** is the compute service for OpenStack used to provision vms and uses different scheduling algorithms to work out where to place virtual machines on available compute.
- **Horizon** is the OpenStack dashboard that users connect to view the status of vms or bare metal servers that are running in a tenant network.
- **Rabbitmq** is the message queue system for OpenStack.
- **Galera** is the database used to store all OpenStack data in the Nova (compute) and neutron (networking) databases holding VM, port, and subnet information.
- **Swift** is the object storage service for OpenStack and can be used as a redundant storage backend that stores replicated copies of objects on multiple servers. Swift is not like traditional block or file-based storage; objects can be any unstructured data.
- **Ironio** is the bare metal provisioning service for OpenStack. Originally, a fork of part of the Nova codebase, it allows provisioning of images on to bare metal servers and uses IPMI and ILO or DRAC interfaces to manage physical hardware.
- **Neutron** is the networking service for OpenStack and contains ML2 and L3 agents and allows configuration of network subnets and routers.

In terms of neutron networking services, neutron architecture is very similar in constructs to AWS.



Useful links covering OpenStack services can be found at:
<http://docs.openstack.org/admin-guide/common/get-started-openstack-services.html>.
<https://www.youtube.com/watch?v=N90ufYN0B6U>

OpenStack tenants

A Project, often referred to in OpenStack as a tenant, gives an isolated view of everything that a team has provisioned in an OpenStack cloud. Different user accounts can then be set up against a Project (tenant) using the keystone identity service, which can be integrated with **Lightweight Directory Access Protocol (LDAP)** or Active Directory to support customizable permission models.

OpenStack neutron

OpenStack neutron performs all the networking functions in OpenStack.

The following network functions are provided by the neutron project in an OpenStack cloud:

- Creating instances (virtual machines) mapped to networks
- Assigning IP addresses using its in-built DHCP service
- DNS entries are applied to instances from named servers
- The assignment of private and Floating IP addresses
- Creating or associating network subnets
- Creating routers
- Applying security groups

OpenStack is set up into its **Modular Layer 2 (ML2)** and **Layer 3 (L3)** agents that are configured on the OpenStack controllers. OpenStack's ML2 plugin allows OpenStack to integrate with switch vendors that use either Open vSwitch or Linux Bridge and acts as an agnostic plugin to switch vendors, so vendors can create plugins, to make their switches OpenStack compatible. The ML2 agent runs on the hypervisor communicating over **Remote Procedure Call (RPC)** to the compute host server.

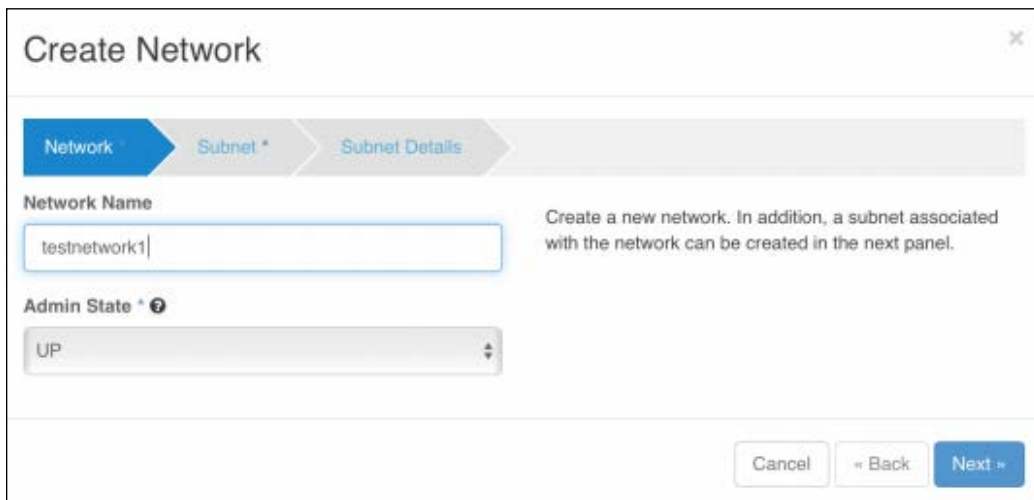
OpenStack compute hosts are typically deployed using a hypervisor that uses Open vSwitch. Most OpenStack vendor distributions use the KVM hypervisor by default in their reference architectures, so this is deployed and configured on each compute host by the chosen OpenStack installer.

Compute hosts in OpenStack are connected to the access layer of the STP 3-tier model, or in modern networks connected to the Leaf switches, with VLANs connected to each individual OpenStack compute host. Tenant networks are then used to provide isolation between tenants and use VXLAN and GRE tunneling to connect the layer 2 network.

Open vSwitch runs in kernel space on the KVM hypervisor and looks after firewall rules by using OpenStack security groups that pushes down flow data via OVSDB from the switches. The neutron L3 agent allows OpenStack to route between tenant networks and uses neutron routers, which are deployed within the tenant network to accomplish this, without a neutron router networks are isolated from each other and everything else.

Provisioning OpenStack networks

When setting up simple networking using neutron in a Project (tenant) network, two different networks, an internal network, and an external network will be configured. The internal network will be used for east-west traffic between instances. This is created as shown in the following horizon dashboard with an appropriate **Network Name**:



The screenshot shows the 'Create Network' form in the OpenStack Horizon dashboard. The form has a title bar with 'Create Network' and a close button. Below the title bar is a progress indicator with three steps: 'Network' (active), 'Subnet *', and 'Subnet Details'. The main form area contains two input fields: 'Network Name' with the value 'testnetwork1' and 'Admin State *' with a dropdown menu set to 'UP'. To the right of the 'Network Name' field is a help text: 'Create a new network. In addition, a subnet associated with the network can be created in the next panel.' At the bottom right of the form are three buttons: 'Cancel', '« Back', and 'Next »'.

The **Subnet Name** and subnet range are then specified in the **Subnet** section, as shown in the following screenshot:

The screenshot shows a 'Create Network' dialog box with a progress bar at the top containing three steps: 'Network', 'Subnet', and 'Subnet Details'. The 'Subnet' step is currently active. Below the progress bar, there is a checked checkbox labeled 'Create Subnet'. To the right of this checkbox is a text instruction: 'Create a subnet associated with the new network, in which case "Network Address" must be specified. If you wish to create a network without a subnet, uncheck the "Create Subnet" checkbox.' Below this instruction are several input fields: 'Subnet Name' with the value 'testsubnet1', 'Network Address *' with the value '192.168.0.0/24', 'IP Version *' with a dropdown menu set to 'IPv4', and 'Gateway IP' with the value '192.168.0.1'. At the bottom left, there is an unchecked checkbox labeled 'Disable Gateway'. At the bottom right, there are three buttons: 'Cancel', '« Back', and 'Next »'.

Finally, DHCP is enabled on the network, and any named **Allocation Pools** (specifies only a range of addresses that can be used in a subnet) are optionally configured alongside any named **DNS Name Servers**, as shown below:

The screenshot shows a 'Create Network' dialog box with a progress bar at the top indicating three steps: 'Network', 'Subnet', and 'Subnet Details'. The 'Subnet Details' step is currently active. Below the progress bar, there is a checkbox labeled 'Enable DHCP' which is checked. To the right of this checkbox is the text 'Specify additional attributes for the subnet.'. Below this, there are three text input fields: 'Allocation Pools' containing the IP range '192.168.0.10,192.168.0.20', 'DNS Name Servers' which is empty, and 'Host Routes' which is also empty. At the bottom right of the dialog, there are three buttons: 'Cancel', 'Back', and 'Create'.

An external network will also need to be created to make the internal network accessible from outside of OpenStack, when external networks are created by an administrative user, the set **External Network** checkbox needs to be selected, as shown in the next screenshot:

Create Network

Name
testnetworkexternal

Project *
testproject

Provider Network Type * ⓘ
Local

Admin State *
UP

Shared
 External Network

Description:
Create a new network for any project as you need.
Provider specified network can be created. You can specify a physical network type (like Flat, VLAN, GRE, and VXLAN) and its segmentation_id or physical network name for a new virtual network.
In addition, you can create an external network or a shared network by checking the corresponding checkbox.

Cancel Create Network

A router is then created in OpenStack to route packets to the network, as shown below:

Create Router

Router Name *
testrouter1

Admin State
UP

Description:
Creates a router with specified parameters.

Cancel Create Router

The created router will then need to be associated with the networks; this is achieved by adding an interface on the router for the private network, as illustrated in the following screenshot:

The screenshot shows a dialog box titled "Add Interface" with a close button (X) in the top right corner. The dialog is divided into two main sections: input fields on the left and a description on the right.

Input Fields:

- Subnet ***: A dropdown menu showing "testnetwork1: 192.168.0.0/24 (testsubnet1)".
- IP Address (optional) ⓘ**: A text input field containing "192.168.0.1".
- Router Name ***: A text input field containing "testrouter1".
- Router ID ***: A text input field containing "fda6f239-fe71-43b5-86a2-45604a52e90a".

Description:

You can connect a specified subnet to the router.

The default IP address of the interface created is a gateway of the selected subnet. You can specify another IP address of the interface here. You must select a subnet to which the specified IP address belongs to from the above list.

Buttons: "Cancel" and "Add interface".

The **External Network** that was created then needs to be set as the router's gateway, as per the following screenshot:

The screenshot shows a dialog box titled "Set Gateway" with a close button (X) in the top right corner. The dialog is divided into two main sections: input fields on the left and a description on the right.

Input Fields:

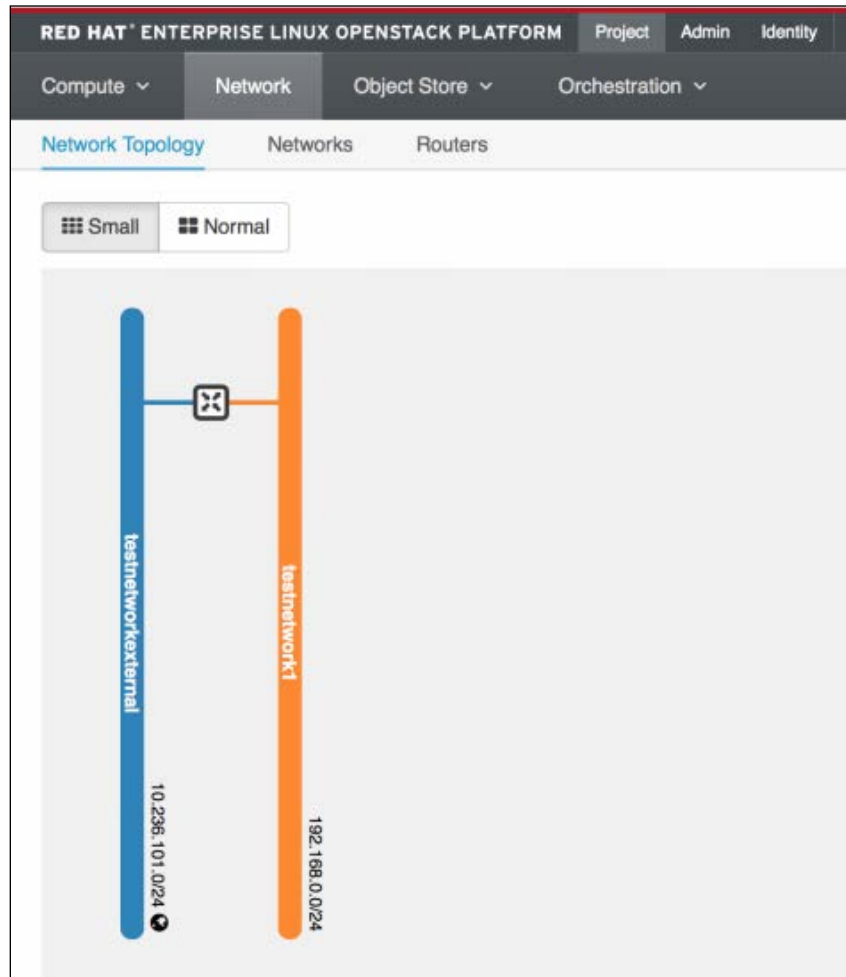
- External Network ***: A dropdown menu showing "testnetworkexternal".
- Router Name ***: A text input field containing "testrouter1".
- Router ID ***: A text input field containing "fda6f239-fe71-43b5-86a2-45604a52e90a".

Description:

You can connect a specified external network to the router. The external network is regarded as a default route of the router and the router acts as a gateway for external connectivity.

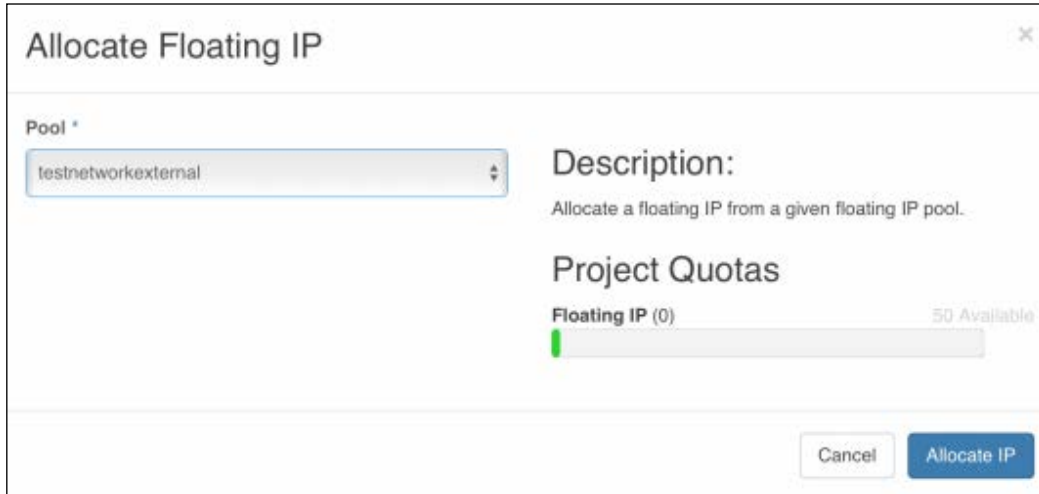
Buttons: "Cancel" and "Set Gateway".

This then completes the network setup; the final configuration for the internal and external network is displayed below, which shows one router connected to an internal and external network:



In OpenStack, instances are provisioned onto the internal private network by selecting the private network NIC when deploying instances. OpenStack has the convention of assigning pools of public IPs (floating IP) addresses from an external network for instances that need to be externally routable outside of OpenStack.

To set up a set of floating IP addresses, an OpenStack administrator will set up an allocation pool using the external network from an external network, as shown in the following screenshot:



Allocate Floating IP [X]

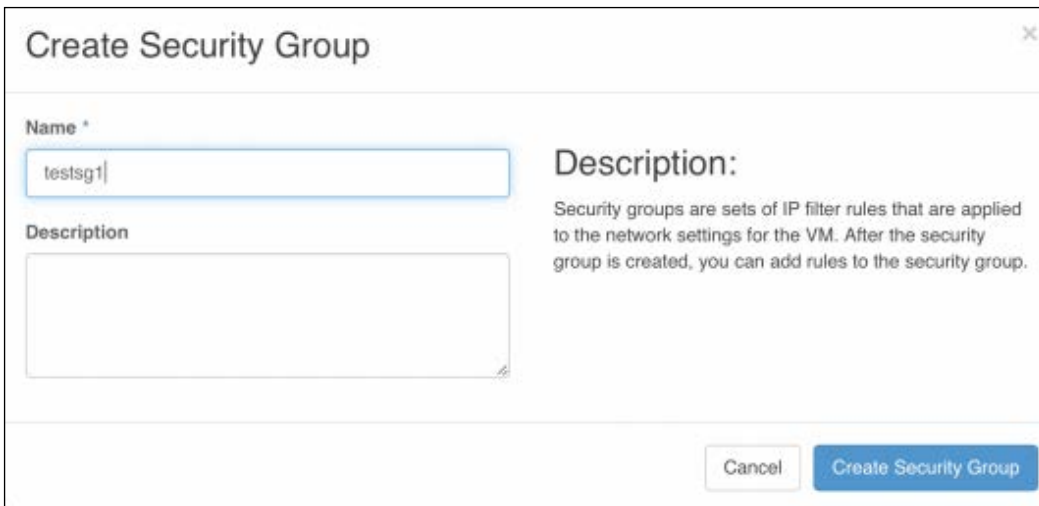
Pool *
testnetworkexternal

Description:
Allocate a floating IP from a given floating IP pool.

Project Quotas
Floating IP (0) 50 Available

Cancel Allocate IP

OpenStack like AWS, uses security groups to set up firewall rules between instances. Unlike AWS, OpenStack supports both ingress and egress ACL rules, whereas AWS allows all outbound communication, OpenStack can deal with both ingress and egress rules. Bespoke security groups are created to group ACL rules as shown below



Create Security Group [X]

Name *
testsg1

Description
Security groups are sets of IP filter rules that are applied to the network settings for the VM. After the security group is created, you can add rules to the security group.

Cancel Create Security Group

Ingress and Rules can then be created against a security group. **SSH** access is configured as an ACL rule against the parent security group, which is pushed down to Open VSwitch into kernel space on each hypervisor, as seen in the next screenshot:

Add Rule

Rule *
SSH

Remote * ⓘ
CIDR

CIDR ⓘ
0.0.0.0/0

Description:

Rules define which traffic is allowed to instances assigned to the security group. A security group rule consists of three main parts:

Rule: You can specify the desired rule template or use custom rules, the options are Custom TCP Rule, Custom UDP Rule, or Custom ICMP Rule.

Open Port/Port Range: For TCP and UDP rules you may choose to open either a single port or a range of ports. Selecting the "Port Range" option will provide you with space to provide both the starting and ending ports for the range. For ICMP rules you instead specify an ICMP type and code in the spaces provided.

Remote: You must specify the source of the traffic to be allowed via this rule. You may do so either in the form of an IP address block (CIDR) or via a source group (Security Group). Selecting a security group as the source will allow any other instance in that security group access to any other instance via this rule.

Add

Once the Project (tenant) has two networks, one internal and one external, and an appropriate security group has been configured, instances are ready to be launched on the private network.

An instance is launched by selecting **Launch Instance** in horizon and setting the following parameters:

- **Availability Zone**
- **Instance Name**
- **Flavor** (CPU, RAM, and disk space)

- **Image Name** (base operating system)

Launch Instance

Project & User * | **Details *** | Access & Security | Networking * | Post-Creation

Advanced Options

Availability Zone: nova

Instance Name *: testinstance1

Flavor * ⓘ: m1.testflavor

Instance Count * ⓘ: 1

Instance Boot Source * ⓘ: Boot from Image

Image Name *: rhel7_base (978.7 MB)

Specify the details for launching an instance. The chart below shows the resources used by this project in relation to the project's quotas.

Flavor Details

Name	m1.testflavor
VCPUs	2
Root Disk	120 GB
Ephemeral Disk	10 GB
Total Disk	130 GB
RAM	8,192 MB

Project Limits

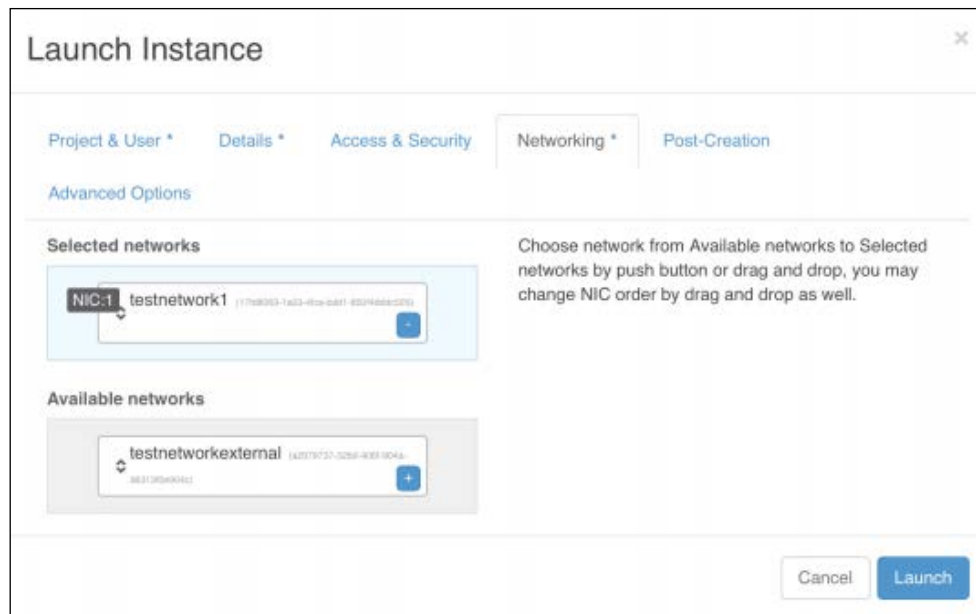
Number of Instances: 0 of 10 Used

Number of VCPUs: 0 of 20 Used

Total RAM: 0 of 51,200 MB Used

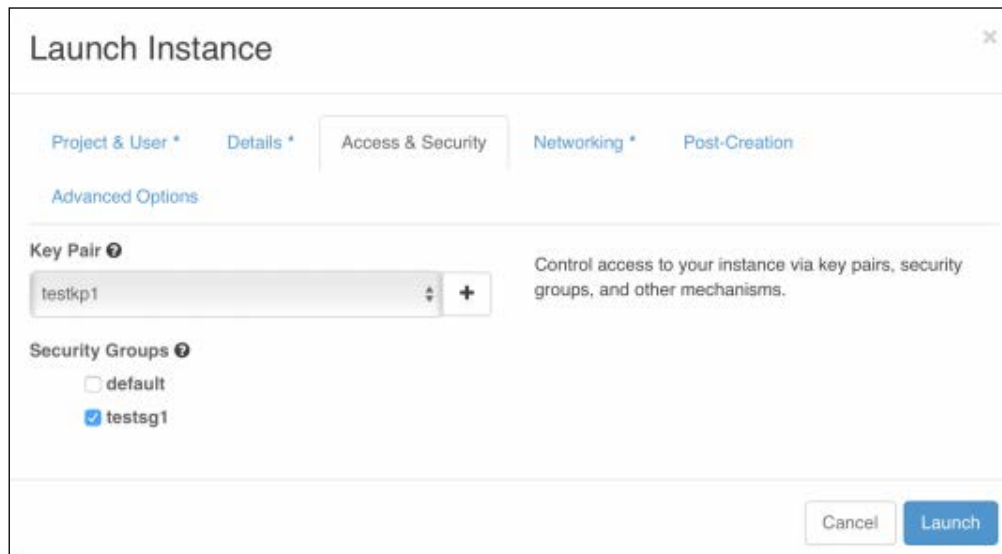
Cancel Launch

The private network is then selected as the NIC for the instance under the **Networking** tab:

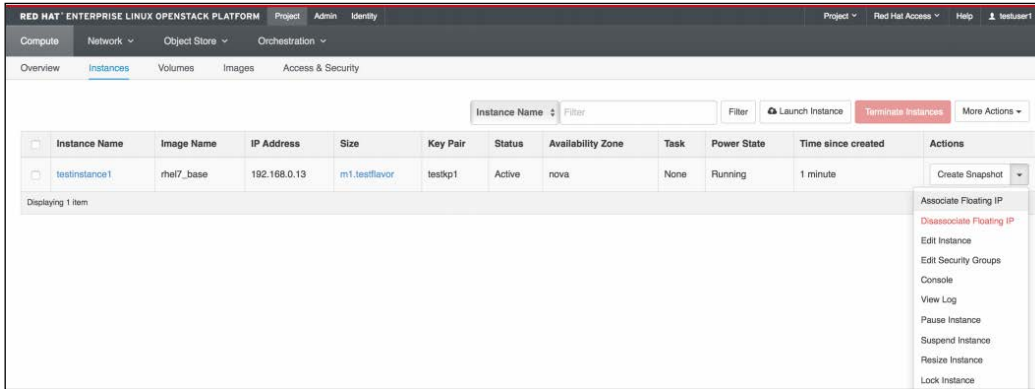


This will mean that when the instance is launched, it will use OpenStack's internal DHCP service to pick an available IP address from the allocated subnet range.

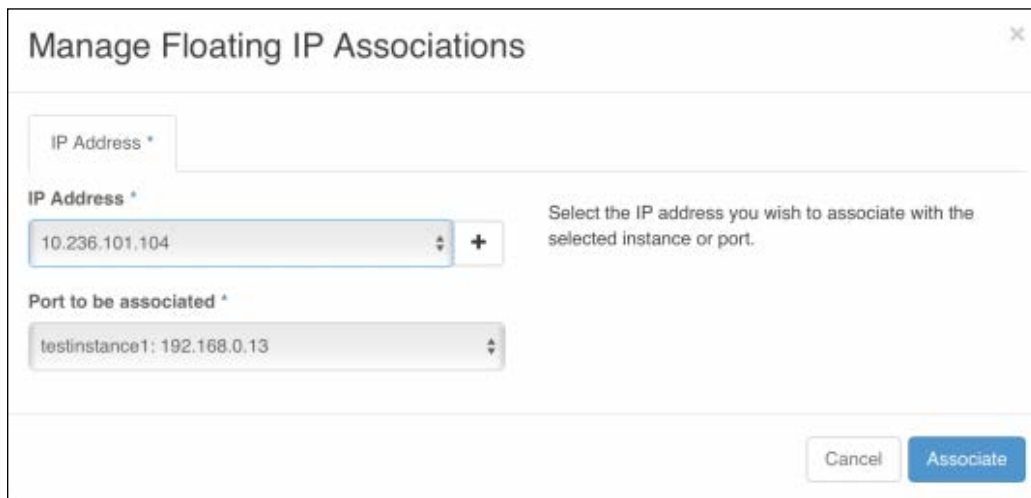
A security group should also be selected to govern the ACL rules for the instance; in this instance, the `testsg1` security group is selected as shown in the following screenshot:



Once the instance has been provisioned, a floating IP address can be associated from the external network:



A floating IP address from the external network floating IP address pool is then selected and associated with the instance:



The floating IP addresses NATs OpenStack instances that are deployed on the internal public IP address to the external network's floating IP address, which will allow the instance to be accessible from outside of OpenStack.

OpenStack regions and availability zones

OpenStack like AWS, as seen on instance creation, also utilizes regions and availability zones. Compute hosts in OpenStack (hypervisors) can be assigned to different availability zones.

An availability zone in OpenStack is just a virtual separation of compute resources. In OpenStack, an availability zone can be further segmented into host aggregates. It is important to note that a compute host can be assigned to only one availability zone, but can be a part of multiple host aggregates in that same availability zone.

Nova uses a concept named **nova scheduler rules**, which dictates the placement of instances on compute hosts at provisioning time. A simple example of a nova scheduler rule is the `AvailabilityZoneFilter` filter, which means that if a user selects an availability zone at provisioning time, then the instance will land only on any of the compute instances grouped under that availability zone.

Another example of the `AggregateInstanceExtraSpecsFilter` filter that means that if a custom flavor (CPU, RAM, and disk) is tagged with a key value pair and a host aggregate is tagged with the same key value pair, then if a user deploys with that flavor the `AggregateInstanceExtraSpecsFilter` filter will place all instances on compute hosts under that host aggregate.

These host aggregates can be assigned to specific teams, which means that teams can be selective about which applications they share their compute with and can be used to prevent noisy neighbor syndrome. There is a wide array of filters that can be applied in OpenStack in all sorts of orders to dictate instance scheduling. OpenStack allows cloud operators to create a traditional cloud model with large groups of contended compute to more bespoke use cases where the isolation of compute resources is required for particular application workloads.

The following example shows host aggregates with groups and shows a host aggregate named **1-Host-Aggregate**, grouped under an **Availability Zone** named **DC1** containing two compute hosts (hypervisors), which could be allocated to a particular team:

Name	Availability Zone	Hosts	Metadata
1-Host-Aggregate	DC1	ie1-tools-compute0-3b06.inf.betfair ie1-tools-compute2-3b06.inf.betfair	availability_zone = DC1

OpenStack instance provisioning workflow

When an instance (virtual machine) is provisioned in OpenStack, the following high-level steps are carried out:

- The Nova compute service will issue a request for a new instance (virtual machine) using the image selected from the glance images service
- The nova request may then be queued by **RabbitMQ** before being processed (RabbitMQ allows OpenStack to deal with multiple simultaneous provisioning requests)
- Once the request for a new instance is processed, the request will write a new row into the nova Galera database in the nova database
- Nova will look at the nova scheduler rules defined on the OpenStack controllers and will use those rules to place the instance on an available compute node (KVM hypervisor)
- If an available hypervisor is found that meets the nova scheduler rules, then the provisioning process will begin
- Nova will check whether the image already exists on the matched hypervisor. If it doesn't, the image will be transferred from the hypervisor and booted from local disk
- Nova will issue a neutron request, which will create a new VPort in OpenStack and map it to the neutron network
- The VPort information will then be written to both the nova and neutron databases in Galera to correlate the instance with the network
- Neutron will issue a DHCP request to assign the instance a private IP address from an unallocated IP address from the subnet it has been associated with
- A private IP address will then be assigned, and the instance will start to start up on the private network
- The neutron metadata service will then be contacted to retrieve cloud-init information on boot, which will assign a DNS entry to the instance from the named server, if specified

- Once cloud-init has run, the instance will be ready to use
- Floating IPs can then be assigned to the instance to NAT to external networks to make the instances publicly accessible

OpenStack LBaaS

Like AWS OpenStack also offers a **Load-Balancer-as-a-Service (LBaaS)** option that allows incoming requests to be distributed evenly among designated instances using a **Virtual IP (VIP)**. The features and functionality supported by LBaaS are dependent on the vendor plugin that is used.

Popular LBaaS plugins in OpenStack are:

- Citrix NetScaler
- F5
- HaProxy
- Avi networks

These load balancers all expose varying degrees of features to the OpenStack LBaaS agent. The main driver for utilizing LBaaS on OpenStack is that it allows users to use LBaaS as a broker to the load balancing solution, allowing users to use the OpenStack API or configure the load balancer via the horizon GUI.

LBaaS allows load balancing to be set up within a tenant network in OpenStack. Using LBaaS means that if for any reason a user wishes to use a new load balancer vendor as opposed to their incumbent one; as long as they are using OpenStack LBaaS, it is made much easier. As all calls or administration are being done via the LBaaS APIs or Horizon, no changes would be required to the orchestration scripting required to provision and administrate the load balancer, and they wouldn't be tied into each vendor's custom APIs and the load balancing solution becomes a commodity.

Summary

In this chapter, we have covered some of the basic networking principles that are used in today's modern data centers, with special focus on the AWS and OpenStack cloud technologies which are two of the most popular solutions.

Having read this chapter, you should now be familiar with the difference between Leaf-Spine and Spanning Tree network architectures, it should have demystified AWS networking, and you should now have a basic understanding of how private and public networks can be configured in OpenStack.

In the forthcoming chapters, we will build on these basic networking constructs and look at how they can be programmatically controlled using configuration management tools and used to automate network functions. But first, we will focus on some of the software-defined networking controllers that can be used to extend the capability of OpenStack even further than neutron in the private clouds and some of the feature sets and benefits they bring to ease the pain of managing network operations.



Useful links for Amazon content are:

<https://aws.amazon.com/>

<https://www.youtube.com/watch?v=VgzZHCukwpc>

<https://www.youtube.com/watch?v=jLVPqoV4YjU>

Useful links for OpenStack content are:

https://wiki.openstack.org/wiki/Main_Page

<https://www.youtube.com/watch?v=Qz5gyDenqTI>

<https://www.youtube.com/watch?v=Li0Ed1VEziQ>

2

The Emergence of Software-defined Networking

This chapter will discuss the emergence of open protocols that have helped **Software-defined Networking (SDN)** solutions. It will focus specifically on the Nuage VSP SDN solution, which is an SDN platform from Nokia, formerly known as Alcatel-Lucent, which allows users to create a virtual overlay network. We will look at some of the scaling benefits and features Nuage VSP provides over and above the out of the box experience from AWS and OpenStack. It will articulate why these networking solutions have become a necessity for notoriously complex private cloud networks, by simplifying networking using software constructs while aiding automation of the network by providing a set of programmable APIs and SDKs.

This chapter will focus on the following topics in detail:

- Why SDN solutions are necessary
- How the Nuage SDN solution works
- The integration of OpenStack with the Nuage VSP Platform
- The Nuage VSP Software-defined object model
- How the Nuage VSP can support Greenfield and Brownfield Projects
- The Nuage VSP Multicast Support

Why SDN solutions are necessary

SDN solutions are necessary as they allow businesses to simplify their network operations, and it also allows them to automate network functions. It fits well with the DevOps initiative and the need to make network operations more agile.

A byproduct of SDN is that it allows network functions to become as accurate and repeatable as creating a new virtual machine on a hypervisor.

SDN solutions from vendors are made up of a centralized controller that is implemented to become the nerve center of the network. SDN controllers rely heavily on **Open vSwitch database (OVSDB)**, which is a programmable, open standard schema which utilizes the OpenFlow protocol, which integrates directly with switches to route packets in the network as well as applying ACL policies to particular virtual machines, physical servers, or containers.

As long as a switch can talk OVSDB and OpenFlow, then it can integrate with common SDN controllers. There are now a wide variety of SDN controllers currently on the market:

- CISCO ACI
<http://www.cisco.com/c/en/us/solutions/data-center-virtualization/application-centric-infrastructure/index.html>
- Nokia Nuage VSP
<http://www.nuagenetworks.net/products/virtualized-services-platform/>
- Juniper Contrail
<http://www.juniper.net/uk/en/products-services/sdn/contrail/>
- VMWare NSX
<http://www.vmware.com/products/nsx.html>
- Open Daylight
<https://www.opendaylight.org/>
- MidoNet Midokura
<http://www.midokura.com/midonet/>

SDN controllers do the following for enterprises:

- Provide an easy-to-use solution for network functions, with the SDN controllers abstracting the network functions from hardware devices and instead expose GUIs and API endpoints that can be programmatically altered to control network operations.
- SDN controllers lend themselves to DevOps models such as self-service network operations for developers, which allow Continuous Delivery of network functions and increased collaboration between teams.

- Provide increased visibility of network configuration as it is described in easy-to-understand software constructs.
- Provide better integration with infrastructure through the use of open networking standards, so this gives companies choice over which switch vendors they integrate.
- Allow the same set of policies in a private datacenter to be applied across private and public clouds. This makes the aim of distributing different workloads into different cloud providers a reality and makes security governance of hybrid clouds much easier for security teams.

The emergence of AWS undoubtedly influenced network vendors to adapt their solutions to be less hardware centric and focus more on a software approach to networking, which, in turn, has simplified network operations and made networking easier to scale.

Vendors have now adopted and implemented open protocols to allow centralized management of network functions and allowed network operators to manage the whole network using an SDN controller.

Software-defined networking is being used by businesses to maximize the performance of their network and create repeatable workflows for network operations in the same way hypervisor virtualization helped infrastructure teams automate server provisioning and management.

However, based on my personal experience, software-defined networking in the private cloud is being used to run OpenStack at massive scale. The continued uptake on OpenStack projects by many major companies, such as Walmart, Ebay, PayPal, Go Daddy, and my company Paddy Power Betfair, means that companies are turning to SDN solutions to allow them to meet necessary scaling targets and simplifying network operations.

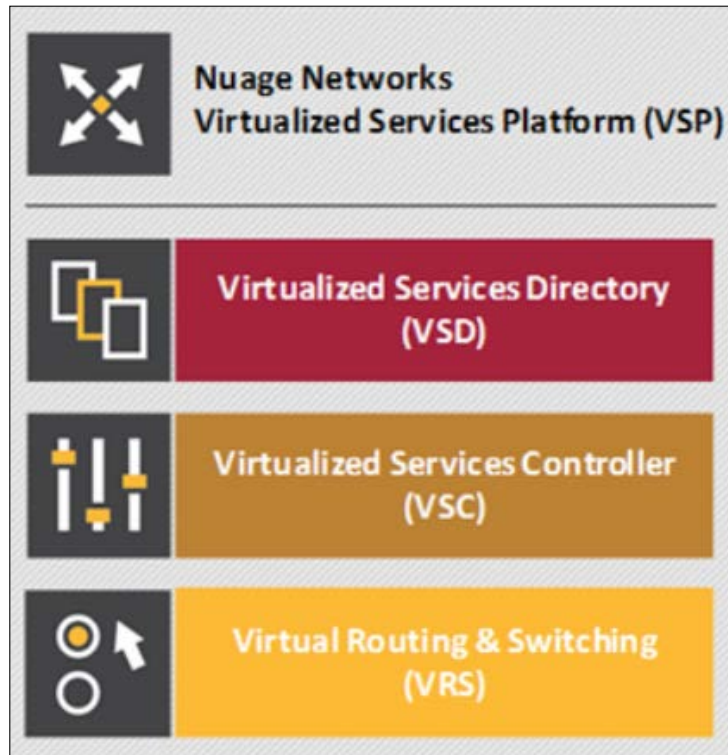
How the Nuage SDN solution works

One of the market-leading SDN solutions is the Nuage SDN (VSP) platform, which is Nokia's SDN solution (formerly Alcatel Lucent), so we will explore how this market-leading SDN solution works.

The Nuage VSP platform comprises three main components – the VSD, VSC, and VRS.

- **Virtualized Service Directory (VSD):** This is the policy engine for the overall platform, and it provides a graphical user interface and exposes a restful API for network engineers to use and interact with network functions.

- **Virtualized Service Controller (VSC):** This is the SDN controller for Nuage, and it uses OpenFlow and OVSDB management protocol to distribute switching and routing information to hypervisors, bare metal servers, or containers.
- **Virtual Routing and Switching (VRS):** This is Nuage's customized version of Open vSwitch, which is installed on compute nodes (hypervisors).



The Nuage VSP can integrate with OpenStack, CloudStack, and VMWare private cloud platforms or public cloud solutions such as AWS. Nuage creates an overlay network that has the ability to secure virtual machines, bare metal servers, and containers in an isolated tenant network, so it is highly flexible depending on what kind of workload needs to be deployed.

Virtual machines, of course, are deployed as a virtual abstraction on top of physical hypervisors, where containers can run on top of virtual machines or physical servers. Containers are used to isolate particular processes or resources using Linux namespaces and control groups, which divide resources at operating system level, so the networking requirements for virtual machines and containers are very different. Containers are used because they are portable and can run on either virtual machines or bare metal (physical) servers. Another advantage is they are encapsulated by the Linux operating system and multiple containers can run on a virtual machine or physical server.

Nuage also supports multicast between tenant networks by routing multicast traffic via hypervisors on the underlay network via hypervisors or physical machines and flooding it to specific virtual or physical machines within a tenant network, which has been somewhat of an issue with cloud solutions, but Nuage has a solution to that particular problem.

The Nuage VSPs SDN Controller (VSC) integrates with switches using OVSDB via hardware VTEPs exposed by switches at the access or leaf layer of the network. VSCs are deployed redundantly and communicate with each other with **Multipath Border Gate Protocol (MP-BGP)** and program VXLAN encapsulation to the switches as they are hardware VTEP aware.

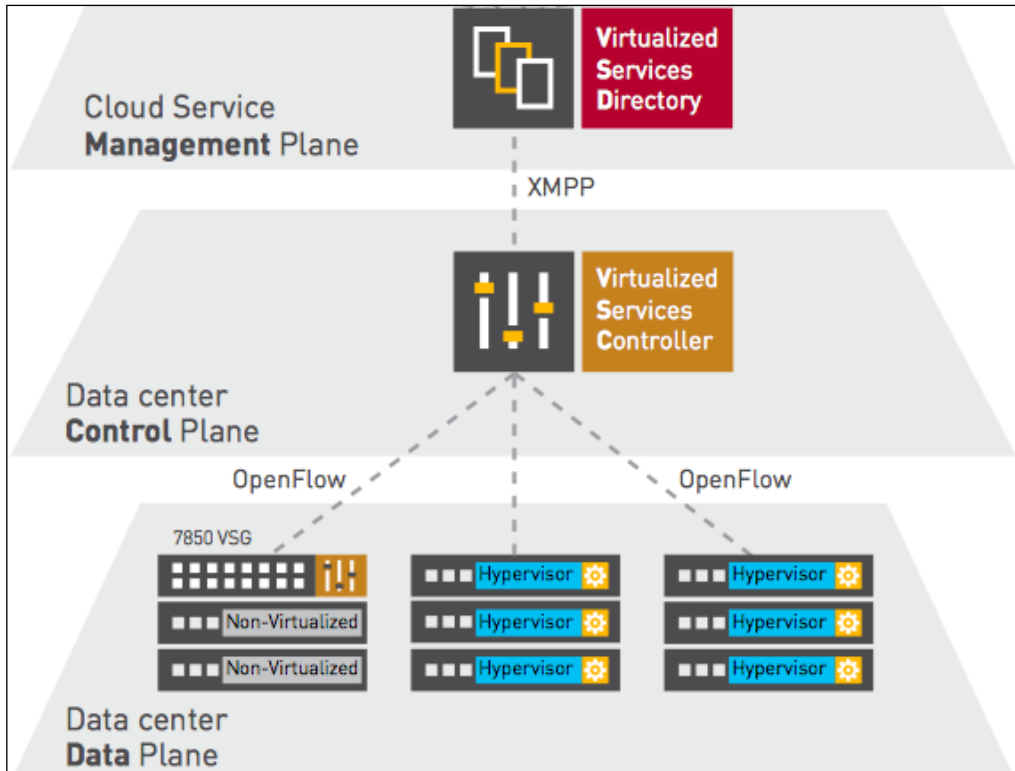
The VSD component is set up in an active cluster containing three VSD servers, which are load balanced using a viable load balancer solution. The load balancer provides a **Virtual IP (VIP)**, which load balances three VSDs servers in round-robin mode.

The VSDs VIP exposes the graphical user interface for the Nuage VSP platform and API entry point to programmatically control the overlay network using REST calls. Any operation carried out on the Nuage VSD GUI initiates a REST API call to the VSD, so both the GUI and the REST API are carrying out identical programmatic calls and all operations are exposed via the REST API.

The Nuage VSD governs layer 2 and 3 domains, zones, subnets, and ACL policies. The VSD communicates policy information to the VSC using **Extensible Messaging and Presence Protocol (XMPP)**, and the VSC uses OpenFlow to push down flow information to a customized version of Open vSwitch (VRS) on the compute hosts (hypervisor) to create firewall policy for applications.

The Nuage VSP allows bare metal servers to be connected to overlay networks too by pushing down OpenFlow Data to the **Virtualized Services Gateway (VSG)** and leaking routing information into the overlay network.

An overview of the VSP platform protocol integration can be found in the following figure:



Integrating OpenStack with the Nuage VSP platform

Private data center networks can be very complex, so using vanilla OpenStack neutron to meet all use cases may not provide all the features that are required. It is important to note that the features in neutron are maturing very quickly with every new OpenStack release, so neutron is likely as feature rich as dedicated SDN controllers in the future.

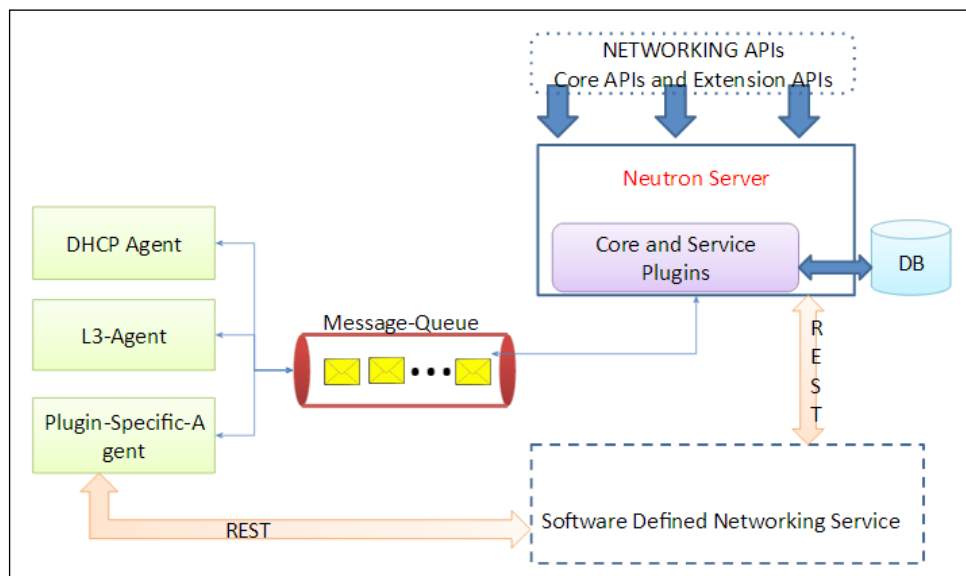
Neutron lends itself to integration with SDN controllers by providing a REST API extension, so SDN controllers can easily be used to extend the base networking functions provided by neutron if required to provide a very rich set of networking features.

The use of SDN solutions have helped OpenStack to scale massively, as it moves the networking aspect of OpenStack away from the centralized layer 3 agent and instead requests are moved to the dedicated SDN controllers with distributed firewalling.

This means that one OpenStack cloud can potentially scale the amount of compute instances that are supported horizontally, without having to worry about bottlenecks or scaling issues associated with the current neutron network architecture.

OpenStack is one of the most popular private cloud solutions, and the Nuage VSP platform integrates with OpenStack using the Nuage plug-in. The Nuage plugin is installed on each of the **Highly Available (HA)** OpenStack controllers.

The Neutron ML2 and L3 agents are both switched off on the controllers in favor of the Nuage plugin. The following image shows the architecture for OpenStack neutrons SDN controller framework with SDN controllers communicating with OpenStack neutron via REST API calls:

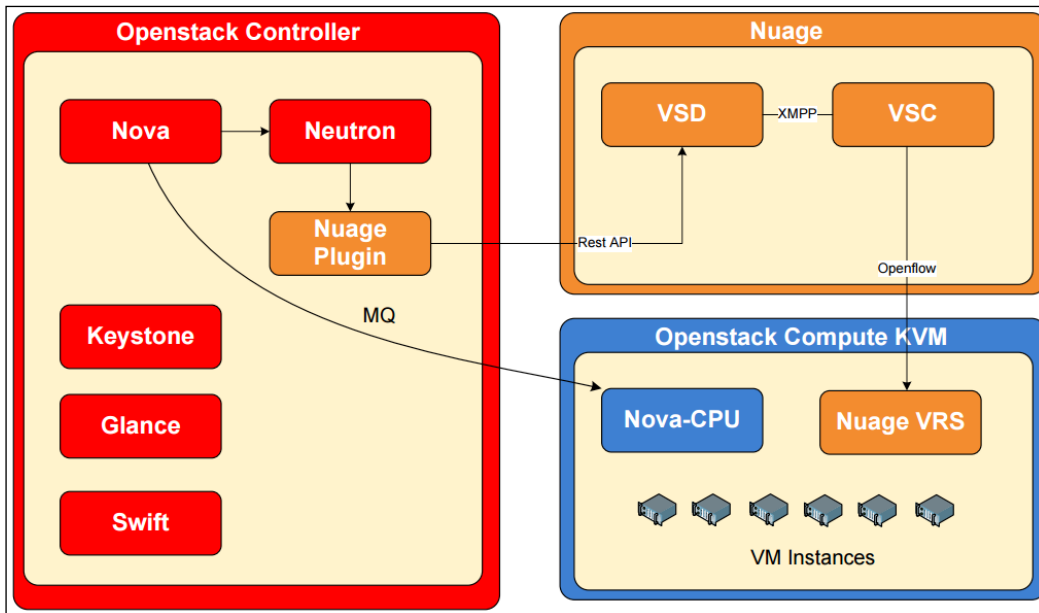


The VSD, which is the Nuage policy engine, integrates with OpenStack by setting up a net partition that can be used to map one Nuage VSP to an OpenStack cloud and communicates with neutron using REST API calls.

Multiple instances of OpenStack can be mapped to a single Nuage VSP via the use of net partitions. Net partitions are a way of telling the Nuage VSP platform, which OpenStack instance to map its subnets to and wait for VPort commands to be issued, which signify that the OpenStack nova compute service, has provisioned a virtual machine instance that needs to be governed by Nuage ACL policies.

When Nuage VSP is integrated with OpenStack, OpenStack vendor installers need to either support Nuage natively or the installer will need to be customized slightly to install the Nuage plugin on OpenStack controllers. The Nuage version of OpenvSwitch (VRS) also needs to be installed on each compute node (hypervisor) that is deployed within an OpenStack cloud.

The Nuage plugin integrates with the OpenStack Controllers and KVM compute using the following workflow:



When a neutron command is issued to OpenStack, the Nuage plugin uses **REST API** calls to communicate with the **Nuage VSD** to say that a new network has been created or a new VPort on that network has been created, this is possible due to neutrons SDN controller pluggable **REST API** architecture.

The **Nuage VSD** policy engine then communicates with the **VSC** to push flow data using XMPP. The **VSC** (SDN Controller) then administers flow data (OpenFlow) to the **Nuage VRS** (Open vSwitch), and the **Nuage VRS** secures OpenStack virtual machines or physical servers with the predefined firewall policies.

Firewall policies can either be OpenStack Security Groups or Nuage ACL rules depending if OpenStack managed mode or Nuage VSD managed mode are selected.

Nuage or OpenStack managed networks

The Nuage OpenStack plugin can be used in two modes of operation to manage networks that are provisioned in OpenStack:

- Nuage VSD-managed mode
- OpenStack-managed mode

Nuage VSD-managed mode allows Nuage to become the master of network provisioning; this allows Nuage VSP platform to provide a rich feature set to manage networks within an OpenStack environment. Network functions are provisioned directly via the VSD using the Nuage REST API by the GUI or direct API calls and mapped one-to-one with OpenStack subnets.

The alternative mode of operation is the OpenStack-managed mode, which requires no direct provisioning on VSD. All commands are issued via neutron; however, functionality is limited to the commands that OpenStack neutron supports.

All networks that are created in Nuage are replicated in OpenStack in a one-to-one mapping with the Nuage VSD being the master in VSD-managed mode, whereas OpenStack neutron is the master of configuration in OpenStack-managed mode.

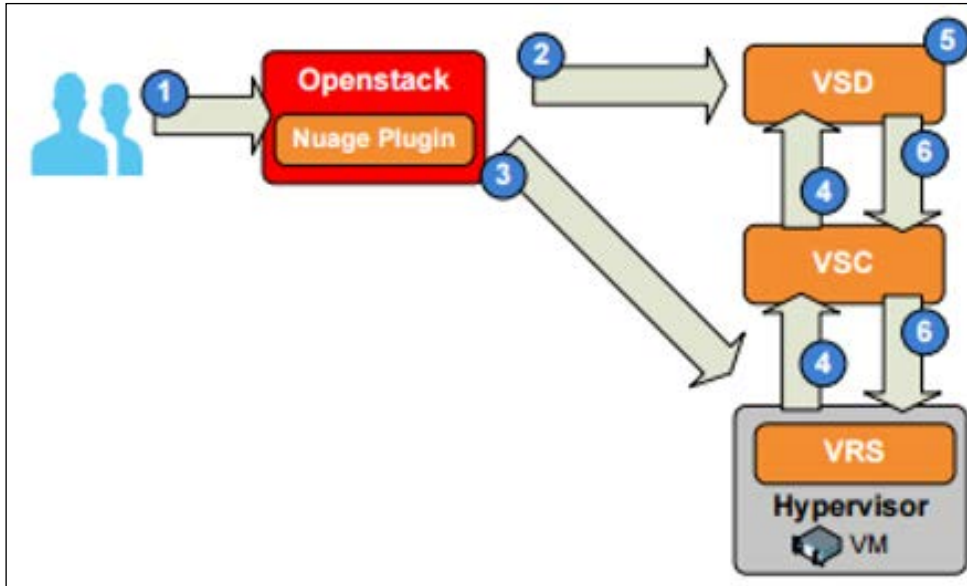
In OpenStack-managed mode, all ACL rules are governed by OpenStack Security Groups, whereas in VSD-managed mode, ACL rules are held instead of the Nuage VSD with security groups disabled.

Nuage integrates with OpenStack by setting up a net partition. Using net partitions, one Nuage VSP Platform can be mapped to multiple instances of OpenStack. Net partitions are a way of mapping an OpenStack cloud to a Nuage organization entity.

Using the Nuage VSP platform with an organization named *Company*, whenever a subnet is created under the organization, it is subsequently assigned a unique `nuage_subnet_uuid` on creation. In order to map the organization and Nuage subnet to OpenStack neutron, the following command is issued:

```
neutron subnet-create "Subnet Application1" 10.102.144.0/24 --nuagenet
nuage_subnet_uuid --net-partition "Company" --name "Subnet Application1"
```

Once a net partition has been established by the Nuage VSP Platform and OpenStack, the firewall policies are secured at the compute host (hypervisor) using the Nuage VRS. The following workflow is triggered when a new instance is created on a Nuage-managed subnet:

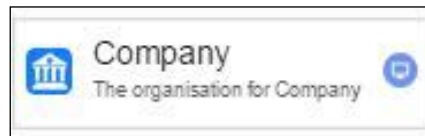


The Nuage VSP software-defined object model

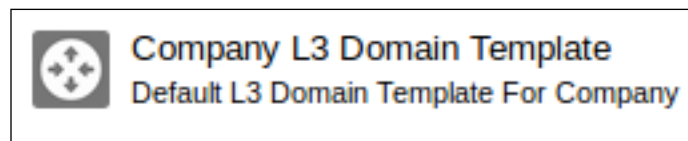
As Nuage creates the overlay network in software, it needs to have a simple object model to allow network operators to manage it. The Nuage VSP software-defined object model provides a graphical hierarchy of the network meaning that the structure of the overlay can be easily viewed and audited.

Object model overview

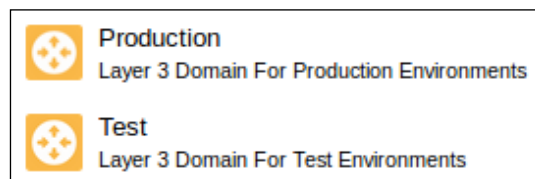
- **Organization:** This governs all Layer 3 domains.



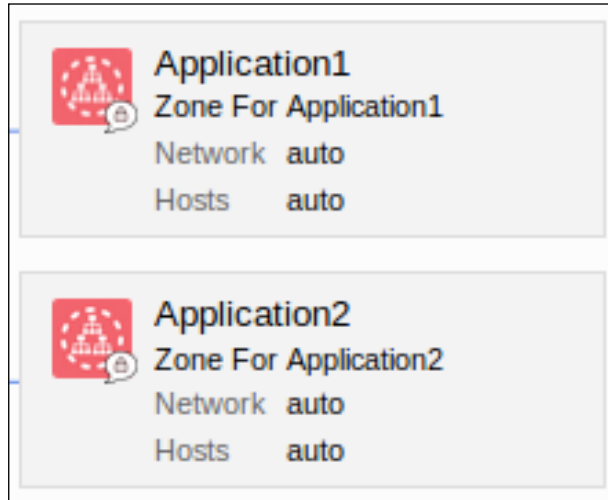
- **Layer 3 domain Template:** A layer 3 domain template is required before child layer 3 domains are created. The layer 3 domain template is used to govern overarching default policies that will be propagated to all child layer 3 domains. If a layer 3 domain template is updated at template level, then the update will be implemented on all layer 3 domains that have been created underneath it immediately.



- **Layer 3 domain:** This can be used to segment different environments, so users cannot hop from subnets deployed under a layer 3 Test domain to an adjacent layer 3 Production domain.

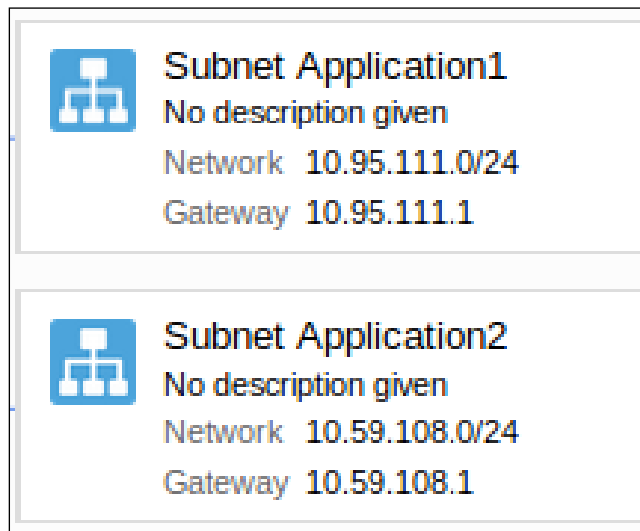


- **Zones:** A zone segments firewall policies at application level, so each microservice application can have its own zone and associated Ingress and Egress policy per layer 3 domain.



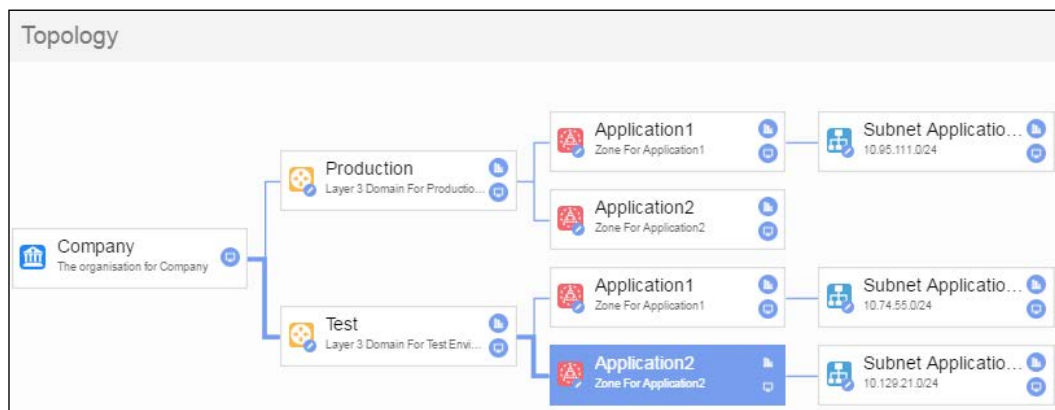
- **Layer 3 Subnet:** This is where VMs or bare metal servers that are deployed to.

In this example, we see **Subnet Application1** and **Subnet Application2**, as follows:



The hierarchy in Nuage VSD is shown below:

- One organization has been created named **Company**.
- Two layer 3 domains named **Test** and **Production** have been created underneath the Company.
- The **Test** layer 3 domain has a zone for **Application1** and **Application2** with 1 child subnet underneath the **Application1** and **Application2** zones.
- The **Production** layer 3 domain has a zone for **Application1** and **Application2** with 1 child subnet underneath the **Application1** only. **Application2** zone does not have a child subnet yet.



For security and compliance purposes, demonstrating to security auditors segmentation between **Development** and **Production** environments is very important. Frequently, **Development** environments do not have the same stringent production controls applied to them. Production applications can be secured using the convention of least privilege possible, to minimize access and reduce the probability of a security breach.

The Nuage VSP Platform can set up segregation between environments using its layer 3 domain template construct. A domain template can be set up with a default **Deny All** policy at Ingress and Egress level. This is given the highest priority of all the policies and will explicitly drop all packets no matter the protocol for inbound and outbound connections, unless explicitly allowed by the policy for that specific application. The default **Deny All** is the bottom policy on the list of ACL rules applied to an application.

The explicit drop on **Egress Security Policy** domain template is shown as the **Bottom policy** as follows:

Edit Egress Security Policy

Name
Default Egress Policy

Description
My Egress Security Policy

Policy Position
Bottom policy

Deploy implicit rules
 Forward IP traffic by default
 Forward non IP traffic by default

Enable this policy Update

The contents of the **Egress Security Policy** are shown with the highest possible priority as follows:

Edit Egress Security Policy Entry

Name

Priority

Enable flow logging

Enable statistics collection

Traffic Type ⚙

Ether Type	<input type="text" value="IPv4 - 0x0800"/>	Source Port	<input type="text" value="*"/>
Protocol	<input type="text" value="TCP - 6"/>	Destination Port	<input type="text" value="*"/>
DSCP Marker	<input type="text" value="Any"/>	Dest. IP Match	<input type="text" value="IP Address"/>

Traffic Path

Origin Network	Destination Location
<input type="text" value="Any"/> ★ Any From anywhere	<input type="text" value="Any"/> ★ Any From anywhere

Traffic Management

Action

Likewise, the explicit drop on Ingress is applied to the domain template as the **Bottom policy**:

Edit Ingress Security Policy Entry

Name

Priority

Enable flow logging

Enable statistics collection

Traffic Type


Ether Type Source Port

Protocol Destination Port

DSCP Marker Source IP Match

Traffic Path

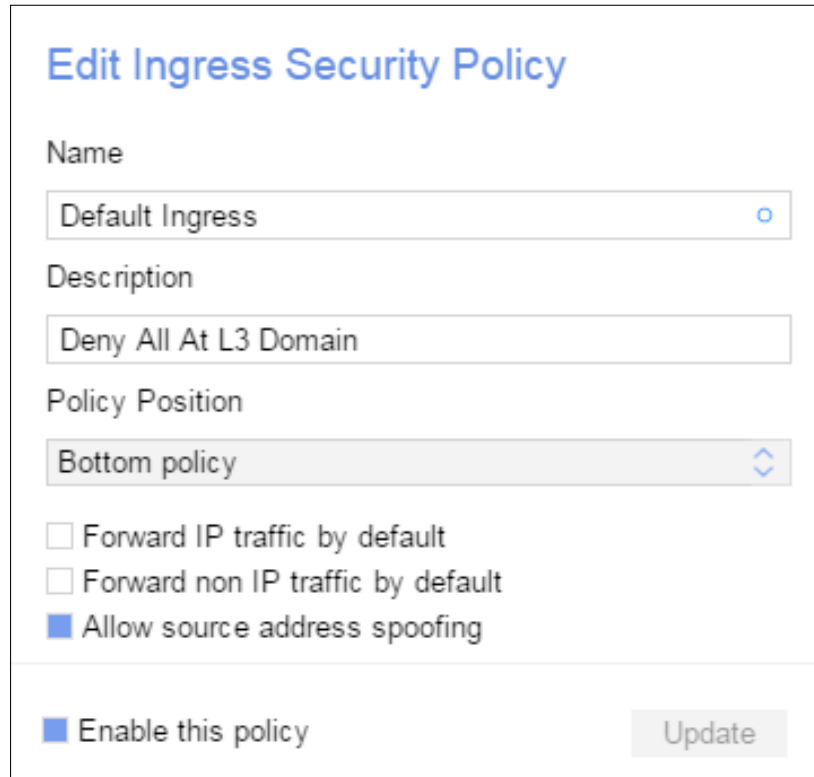
Origin Location Destination Network



Traffic Management

Action

While the explicit drop on the **Ingress Security Policy** on the domain template is shown as follows:



Edit Ingress Security Policy

Name
Default Ingress

Description
Deny All At L3 Domain

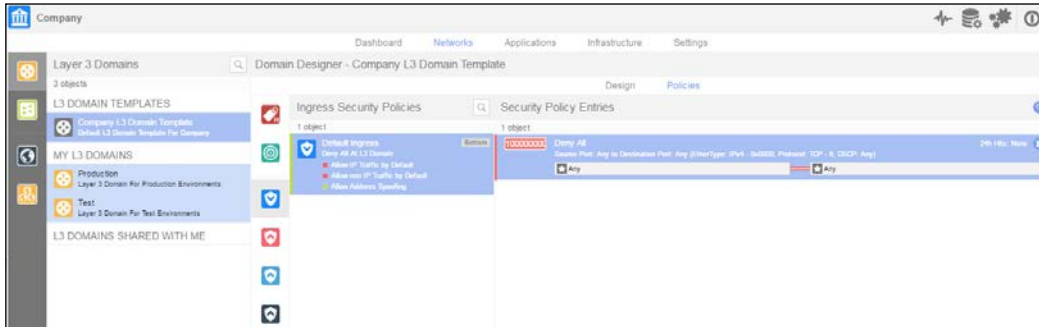
Policy Position
Bottom policy

Forward IP traffic by default
 Forward non IP traffic by default
 Allow source address spoofing

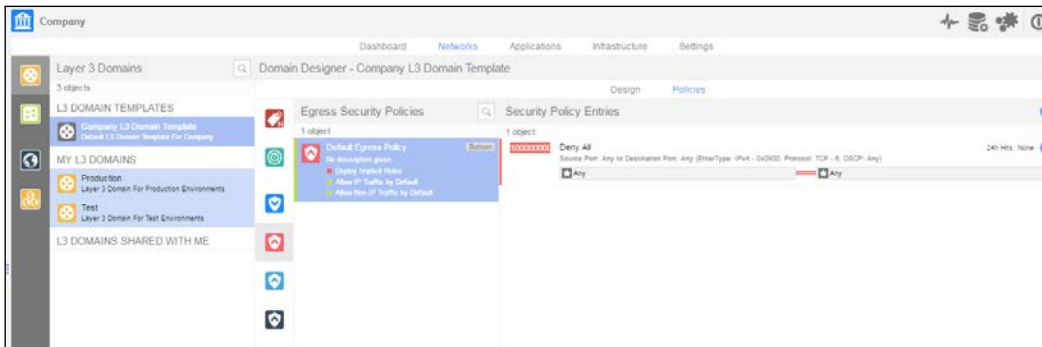
Enable this policy Update

The default Ingress and Egress policies applied to the domain template **Company L3 Domain Template** are illustrated below, which shows the policy applied to all the child layer 3 domains, in this instance, **Production** and **Test**.

The domain template **Company L3 Domain Template** is shown to be linked to the child layer 3 domains **Production** and **Test** showing the inherited Egress policy from the domain template:



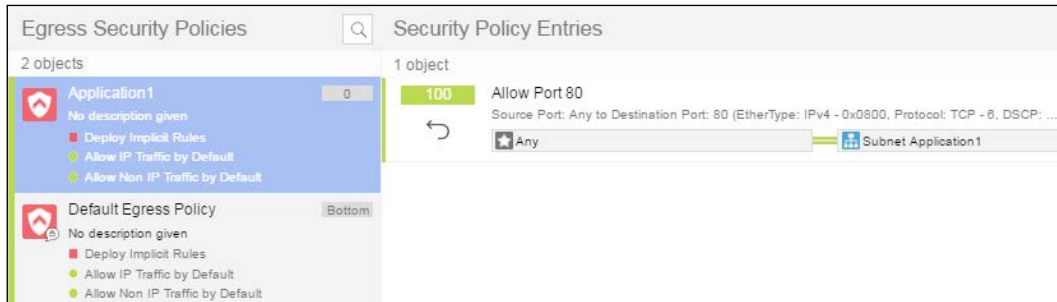
Likewise, the domain template **Company L3 Domain Template** is linked to the child layer 3 domains **Production** and **Test** showing the inherited Ingress policy from the domain template:



It is important to note that, as policies are pushed down to VRS using OpenFlow that ACL rules for Ingress and Egress in Nuage work on the principles:

- **Egress:** This is a packet flowing from VRS to the subnet or zone
- **Ingress:** This is a packet flowing from the subnet or zone to the VRS

As an example, an Egress ACL rule will specify that any Egress traffic coming from VRS from port 80 will be forwarded to **Subnet Application1**:



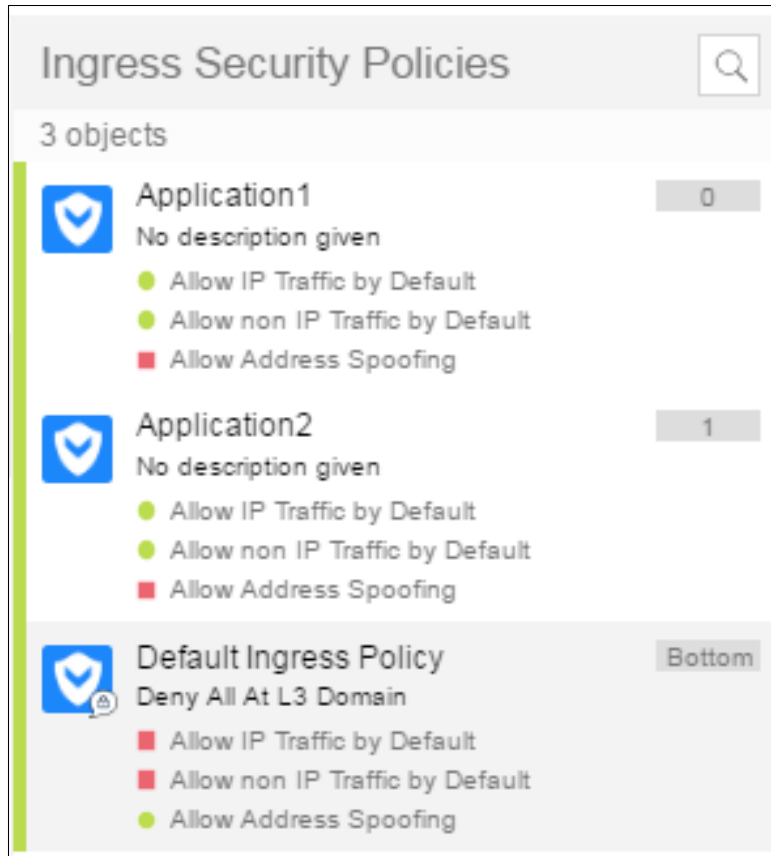
In this example, an Ingress ACL rule will specify that any Ingress traffic can leave **Subnet Application1** on port 80 and will be forwarded to VRS:



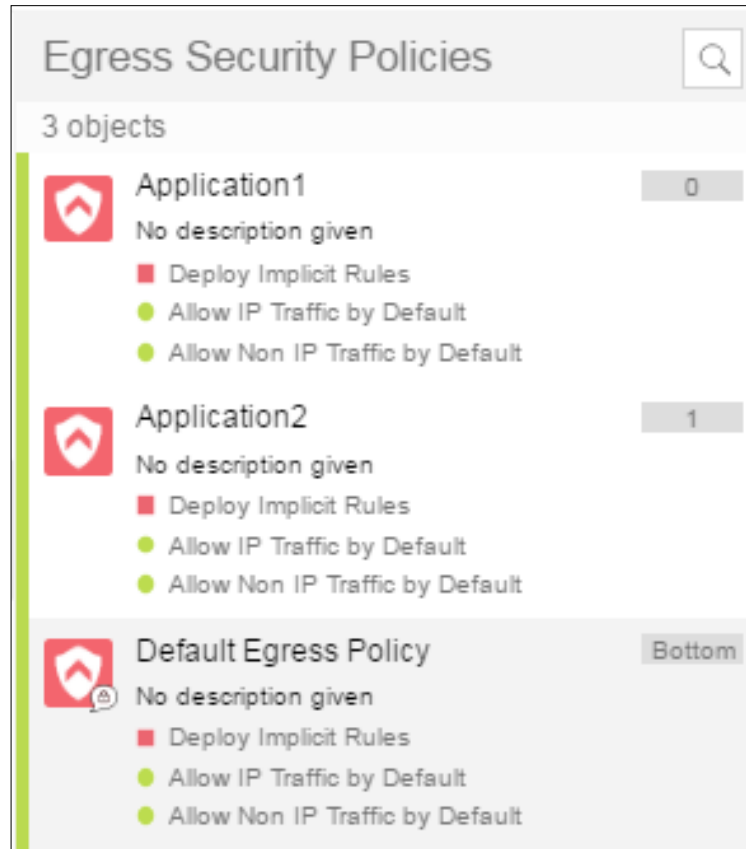
If application owners work on the principle that their layer 3 subnet, where their application is deployed on, is always specified in an ACL rule as either the source or destination in their individual application policy, then ACL rules for an application will only exist in that self-contained policy. If this concept is adhered to, it allows ACL rules for each application to be encapsulated in separate policies, within a layer 3 domain, which, in turn, means that auditing them is much simpler for security teams. It also means that applications support least privilege, meaning only necessary ports are opened so applications can communicate, with an explicit drop applied to anything outside those rules.

Two policies are shown for two applications, **Application1** and **Application2**, which have separate policies for Ingress and Egress, with the **Default Ingress Policy** specifying the explicit drop all for any flows not explicitly allowed.

The **Ingress Security Policies** are shown here:



The **Egress Security Policies** are shown here:



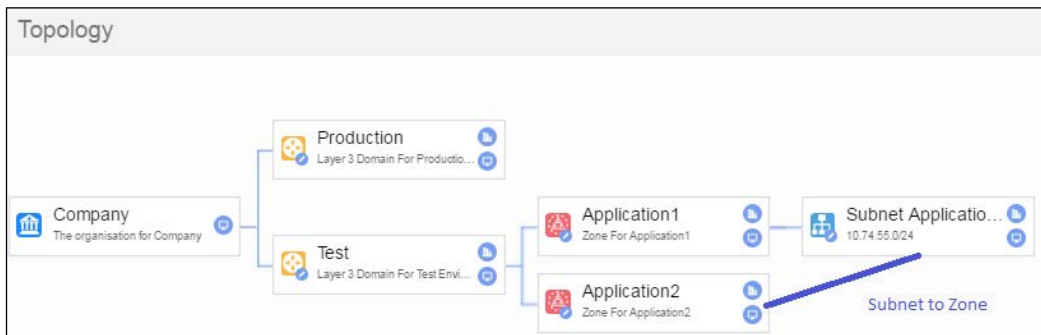
Nuage VSP Platform layer 3 domain templates allow a second level of segmentation using zones, so traditionally networks were split into three zones, where applications would be deployed in the following tiers:

- Frontend
- Business Logic
- Backend

As microservice architectures have grown to prominence, each applications profile doesn't always fit into these three broad profiles. Sometimes, applications can be both a Frontend application and Business Logic too, so where would the microservice application be placed in the traditional three-tiered structure?

Instead of the Frontend, Business Logic and Backend segregation policies that can be applied at zone level, meaningful microsegmentation of applications is possible between each subnet. So how does this translate to Nuage?

If an application wishes to talk to another application it will have an ACL rule that specifies Subnet to Zone communication for east to west communication between applications sitting on adjacent subnets in a layer 3 domain. Nuage allows this by allowing applications to talk Subnet to Zone.



To allow this communication, **Application1** could have an ACL policy to allow **Application2** zone to allow traffic to flow into the subnet on port 22 allowing east to west communication, so no matter how many different subnets are used then **Application1** will always be allowed to communicate with any applications sitting under the **Application2** zone.

Edit Egress Security Policy Entry

Name:

Priority:

Enable flow logging

Enable statistics collection

Traffic Type

Ether Type: Source Port:

Protocol: Destination Port:

DSCP Marker: Dest. IP Match:

Traffic Path

Origin Network: Destination Location:

Zone: Subnet:

Traffic Management

Action:

Create an implicit reflexive rule

In terms of security policies, this allows development and security teams to understand which applications are communicating with each other and the ports they are using by reviewing the application policy.

Egress Security Policies | **Security Policy Entries**

3 objects | 2 objects

Application 1 (0 hits)
No description given
Deploy Implicit Rules
Allow IP Traffic by Default
Allow Non IP Traffic by Default

Application 2 (1 hit)
No description given
Deploy Implicit Rules
Allow IP Traffic by Default
Allow Non IP Traffic by Default

Default Egress Policy (Bottom)
No description given
Deploy Implicit Rules
Allow IP Traffic by Default
Allow Non IP Traffic by Default

Allow Port 80 (100 priority, 24h Hits: None)
Source Port: Any to Destination Port: 80 (EtherType: IPv4 - 0x0800, Protocol: TCP - 6, DSCP: Any)
Any to Subnet Application1

Allow Port 22 Application 2 (200 priority, 24h Hits: None)
Source Port: Any to Destination Port: 22 (EtherType: IPv4 - 0x0800, Protocol: TCP - 6, DSCP: Any)
Application2 to Subnet Application1

How the Nuage VSP platform can support greenfield and brownfield projects

Overlay networks are typically set up as new network (greenfield) sites, but a completely new network in isolation is not useful, unless there is a planned big bang migration of all applications, which means migrating every application from the legacy network to the new network in a single migration.

If instead a staged application migration is chosen, where only a percentage of applications are migrated to the network, then the new overlay network will need to communicate with the legacy network and be required to operate in a brownfield setup.

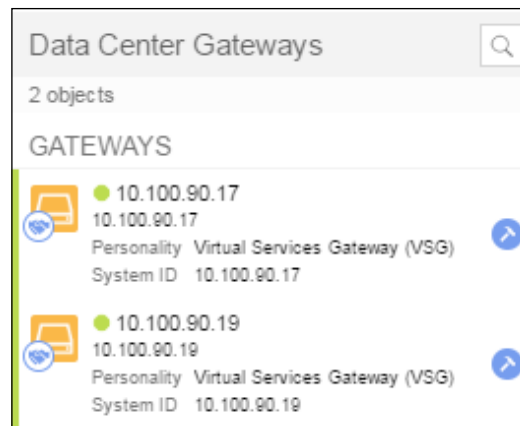
A brownfield setup normally means applications are migrated in stages to the new platform, as opposed to all in one go, which builds confidence in the new network and new technology associated with that network. When moving applications to the new platform, it will typically involve performance testing the migrated applications in the new network, prior to throttling live traffic away from the incumbent legacy network to the migrated application in the new overlay network.

A major requirement for a staged migration is connectivity back to the legacy network for application dependencies that are hosted there. This connectivity is necessary so migrated applications can operate effectively.

The Nuage VSP Platform uses its **Virtualized Service Gateway (VSG)** to provide the connectivity between the new overlay and legacy network. A pair of Nuage VSGs are connected redundantly in virtual chassis mode which connect to interfaces on routers sitting in the legacy network. VSG performs a route table lookup based on the destination IP of a packet coming in on its VLAN from the attached router interface; it then updates the destination MAC with the next hop address and forwards the packet on the corresponding VXLAN segment. All packets are routed from the legacy network to the VSG via an underlay network.

This bridges the new overlay network, and the legacy network with VXLAN terminated on VSG.

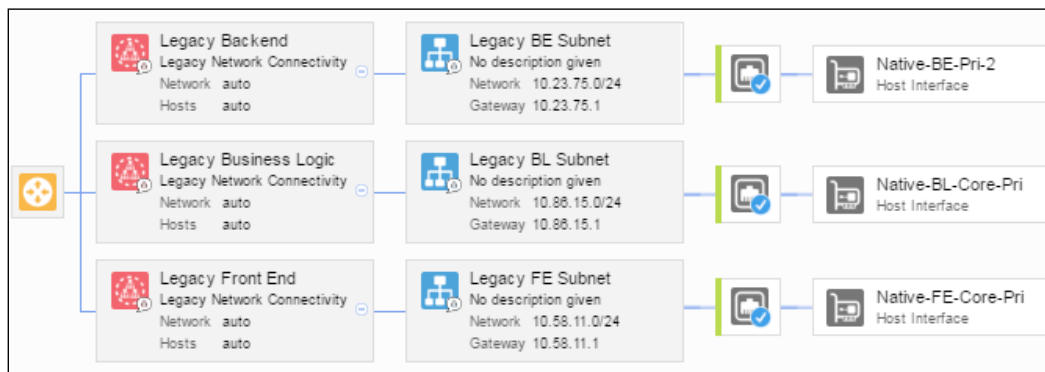
The pair of active VSGs is shown below in Nuage VSD:



The Nuage VSG allows communication with the legacy network by leaking routes to the overlay network. Each VSG will receive and advertise IPv4 routes using a BGP session, this BGP session will be established between VSG, VSC and leaf switch when using a leaf spine topology using iBGP.

The VSG must advertise its local system IP to legacy routers in the legacy network and all routes received from the native network will then be subsequently leaked from the native network via the underlay network into selected layer 3 domains in the overlay.

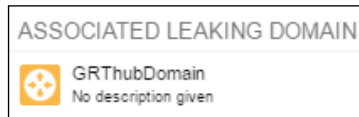
The setup required to leak routes in the Nuage VSP Platform is the creation of a **GRThubDomain** layer 3 domain. In this example, host interfaces are connected into the Frontend, Business Logic, and Backend routers in the legacy network:



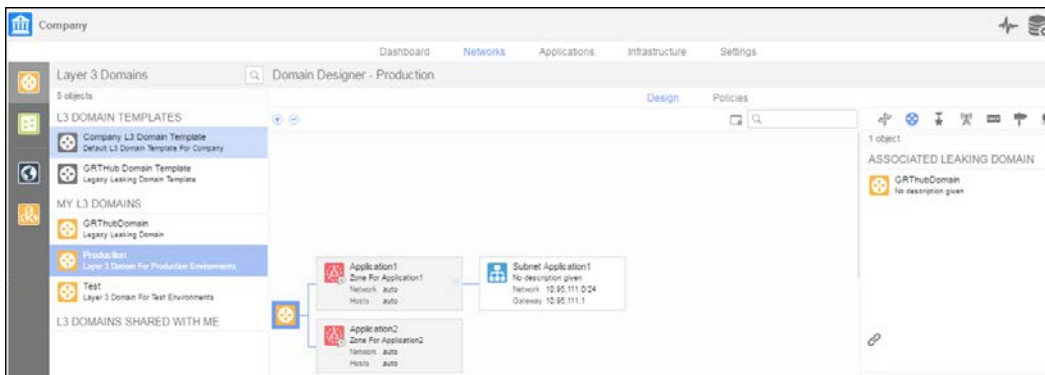
The Nuage VSP platform then allows the newly created **GRThubDomain** to be associated with the **Production** or **Test** layer 3 domains by associating a leaking domain against them.

In the following example, **GRThubDomain** is associated with the **Production** layer 3 domain.

The leaking domain in the Nuage GUI is displayed using the following icon showing a leaking domain named **GRThubDomain**:



The **Production** domain with an associated leaking domain is shown in the Nuage GUI as follows:



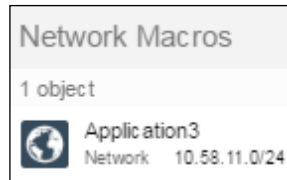
The association of a leaking domain allows the Nuage VSP Platform to leak routes into and from the legacy network through to the new overlay network, meaning applications in the overlay network can communicate with applications in the legacy network, so long as they have appropriate Ingress and Egress ACL policies specified.

The **Test** and **Production** layer 3 domains, as explained before, have a **Deny All** for Ingress and Egress as part of the **Company L3 Domain Template**. So although all routes are leaked into the overlay, they are dropped by the VRS unless explicitly stated otherwise.

The Nuage VSP platform has the ability to apply ACL rules to the routes leaked from the external legacy network by using a concept named **Network Macros**. In the Nuage VSP Platform, a network macro is simply a fancy name for an external network range.

If an application, **Application3** in this instance, resides in the legacy network, and its routing has already been exposed by the **GRThubDomain** leaking domain and leaked into the **Test** layer 3 domain, then a network macro can be set up to describe the range required and isolate connectivity to it using a Nuage ACL rule.

In this instance, the network range 10.58.11.0/24 is where **Application3** resides is part of the Frontend range on the **GRThumbdomain** that is leaked into the overlay network. The **Network Macros** for **Application3** as it would appear in Nuage is shown below:



An Egress ACL policy can then be configured to allow **Application1** to communicate with **Application3** by creating a Network Macros to a subnet ACL rule, which allows **Application3 Network Macros** to connect to **Subnet Application1** on port 8080.

The **Egress Security Policy** to allow communication between **Application3 Network Macros** and **Subnet Application1** on port 8080 is shown here:

New Egress Security Policy Entry

Name

Priority

Enable flow logging

Enable statistics collection

Traffic Type

Ether Type Source Port

Protocol Destination Port

DSCP Marker Dest. IP Match

Traffic Path

Origin Network Destination Location

Application3
10.58.11.0/24

→

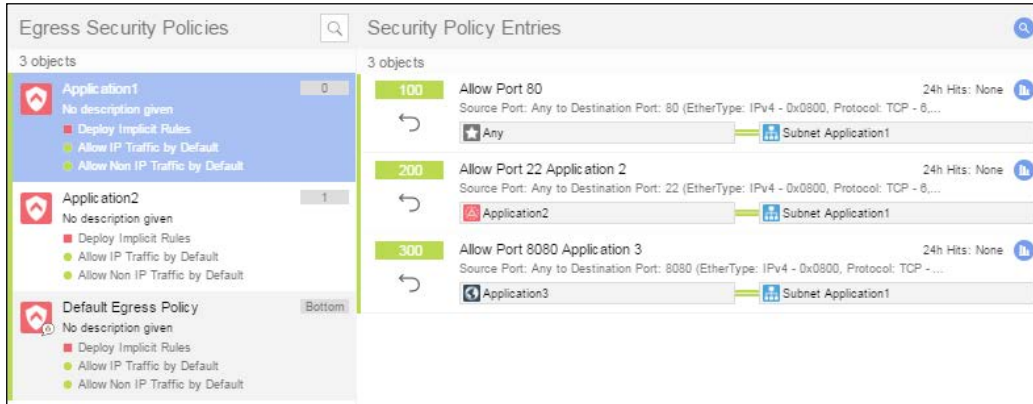
Subnet Application1
No description given

Traffic Management

Action

Create an implicit reflexive rule Create




After creation, the ACL list is updated to show the new **Network Macro** ACL:



This allows the Nuage VSP to lock down policy and the overlay network to only allow specific flow data from the legacy network in the same way it would control ACL policies between subnet or zones that resided within the same layer 3 domain. Network macros can also be used to route between multiple cloud technologies as well as different data centers, so they are a very powerful way of connecting networks and controlling policy between them.

Multiple network macros can be grouped together into a network macro group, which allows multiple ranges to be controlled by one ACL rule. These are then exploded out at the OpenFlow level on VRS at the hypervisor. Nuage currently has a limit of 100 ACL rules per VPort in the 3.x release, so only 100 ACL rules can currently be applied to a single instance (virtual machine), so it is important to be careful when grouping **Network Macros**. This has been increased to 500 ACL rules in the 4.x release of the Nuage platform.

An example of a network macro group can be shown below and then the **Front End Services Network Macro Group** can be used in the Egress ACL rule as opposed to specifying individual policies for **Application3** and **Application4**:

Network Macro Groups	Network Macros
1 object	2 objects
 Front End Services No description given	 Application3 Network 10.58.11.0/24
	 Application4 Network 10.58.12.0/24

The **Egress Security Policy** to allow port 8080 connection between the **Front End Services Network Macro Group** and **Subnet Application1**:

Edit Egress Security Policy Entry

Name:

Priority:

Enable flow logging

Enable statistics collection

Traffic Type


Ether Type: Source Port:


Protocol: Destination Port:

DSCP Marker: Dest. IP Match:

Traffic Path

Origin Network: Destination Location:


Front End Services
 No description given

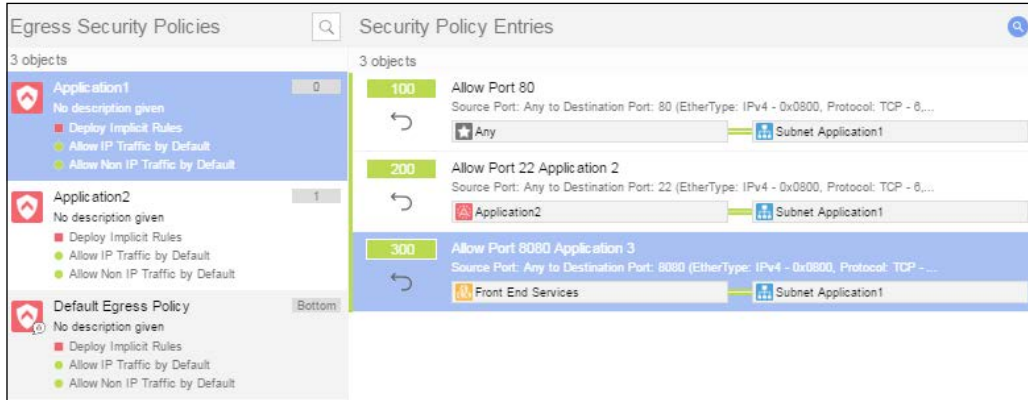

Subnet Application1
 No description given

Traffic Management

Action:

Create an implicit reflexive rule

The applied ACL implementing the **Front End Services** network macro can be found here:



The Nuage VSP multicast support

The Nuage VSP Platform has the ability to route multicast between the following Nuage VSD entities:

- Layer 2 and 3 domains
- Zones
- Subnets
- VPorts attached to VMs

Multicast can be routed into the overlay network, which is a unique feature of the Nuage VSP platform. Multicast traffic is routed into the overlay network in Nuage by configuring dedicated VLANs on the underlay layer 2 network, which are attached to compute nodes. This allows the compute (hypervisors) on the underlay network to use the dedicated VLANs, which are IP'd on a per rack basis, to transmit and receive multicast traffic.

To route multicast traffic across the underlay, the Nuage VRS will duplicate the multicast packets and leak it into the overlay network in a controlled fashion. This is so the overlay network is not flooded with unnecessary multicast traffic, which can cause performance implications to the overlay network if it was not controlled. This makes the Nuage multicast setup highly scalable as it only directs multicast traffic to where it needs to in the overlay network.

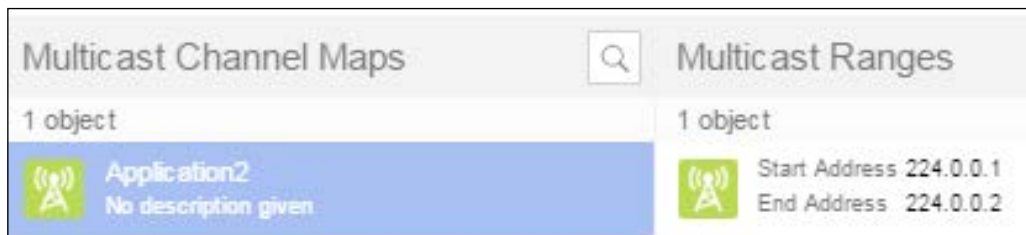
The Nuage VSP uses a dedicated VLAN for multicast send and VLAN for multicast receive on the compute nodes (hypervisor). Each of these VLANs can be configured on each hypervisor in the event applications that have a multicast requirement.

Each of the hypervisors is allocated a VLAN and unique IP address for multicast send and receive, depending on the rack the hypervisors are provisioned on, so they use the associated switches.

Port Channel Maps are the entity used in Nuage VSD to leak multicast from the underlay network to the overlay network. Port Channel Maps are only required if multicast needs to be routed subnet to subnet in the overlay network. If multicast is required in the same subnet, then a **Port Channel Maps** is not required and multicast will work within an isolated layer 3 subnet without having to route the traffic via the VLANs on the hypervisors.

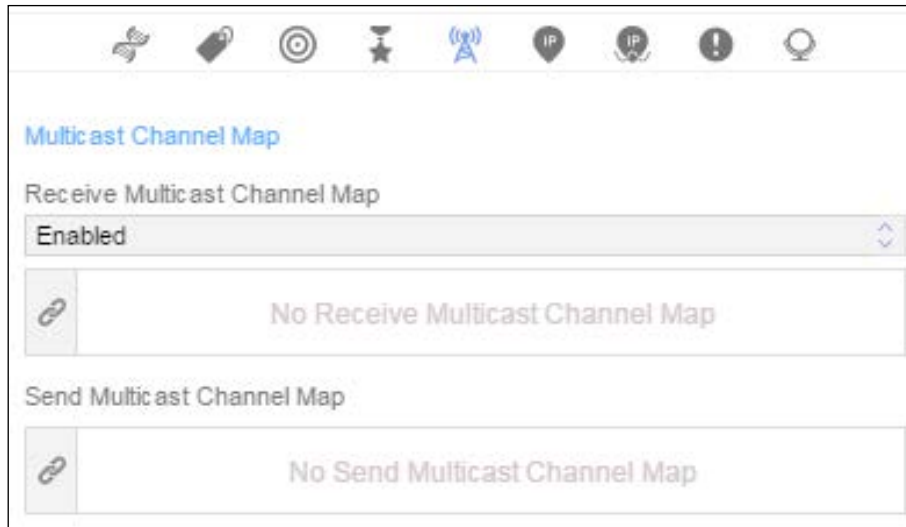
In the following example, a **Multicast Channel Map** is used to create **Multicast Ranges** for **Application2**, which broadcasts multicast. This will route multicast from **Subnet Application2**, via the underlay VLAN on the hypervisor, to the Nuage VRS and then flood it into **Subnet Application1**.

The **Multicast Channel Maps** icon is shown here:



The following scenario describes the workflow an application will go through to route multicast traffic from one layer 3 subnet to another in the overlay.

Application1 will be deployed under a layer 3 domain, under its own zone, in a /26 microsubnet and on that subnet two virtual machines will be attached to two VPorts. Against a VPort in Nuage, a virtual machine can be set up as a sender or receiver of multicast or both:



Application1 in this instance is the sender of multicast and wants to send a multicast stream to **Application2**, which is deployed under a layer 3 domain, under its own zone, in a /26 microsubnet, and on that subnet, one virtual machines will be attached to one VPort.

So a Port Channel Map will need to be set up on **Application1** by associating it with each of the **Application1** VPorts, which lets Nuage know that **Application1** is the multicast sender.

Application2 will have its VPort configured with a Port Channel Map setup, so it can receive multicast.

When Application1s two virtual machines broadcast multicast traffic, Nuage now knows to route the multicast traffic on the matching multicast range specified on the Port Channel Map to the hypervisor that **Application1** is deployed on.

Nuage will transmit multicast across the hypervisor layer 2 domain using the sender VLAN.

Each of the receiver VLANs on each hypervisor's receiver IPs will then pick up the transmission of multicast.

If a Port Channel Map is specified on any of the virtual machines on the hypervisor, matching the **Multicast Ranges** configured, which **Application2** does, then the Nuage VRS will duplicate the multicast packets, leaking them into the overlay network to **Application2**.


This is how Nuage leaks multicast traffic to the overlay network, using the underlay network and sender and receiver VLANs.

Summary

In this chapter, we covered some of the advanced networking features provided by the Nuage VSP SDN solution and also touched upon some of the other SDN solutions that are available on the market. Having read this chapter, you should now be familiar with the Nuage SDN controller and understand the rich set of features an SDN controller can bring to OpenStack and the private cloud.

Given the programmability SDN controllers, AWS, and OpenStack solutions bring, we will now shift focus and look at the cultural changes that are necessary in organizations to make the most of these fantastic technologies. Implementing new technologies without changing operational models is not enough, people and process are key to a successful DevOps model.

The role of the network engineer is undergoing its biggest evolution in years, so businesses cannot simply implement new technology and expect faster delivery without dealing with people and cultural issues. CTOs have a responsibility to set their networking teams up for success by implementing DevOps transformations that include network functions, and network teams also need to learn new skills such as coding to push forward automation using grass root initiatives.

 Useful links on Nuage Networks practical use cases are:
https://www.youtube.com/watch?v=_ZfFbhmiNYo
<https://www.youtube.com/watch?v=aKa2idHhk94>
<https://www.youtube.com/watch?v=OjXII11hYwc>

3

Bringing DevOps to Network Operations

This chapter will switch the focus from technology to people and processes. The DevOps initiative was initially about breaking down silos between development and operations teams and changing companys' operational models. It will highlight methods to unblock IT staff and allow them to work in a more productive fashion, but these mindsets have since been extended to quality assurance testing, security, and now network operations. This chapter will primarily focus on the evolving role of the network engineer, which is changing like the operations engineer before them, and the need for network engineers to learn new skills that will allow network engineers to remain as valuable as they are today as the industry moves towards a completely programmatically controlled operational model.

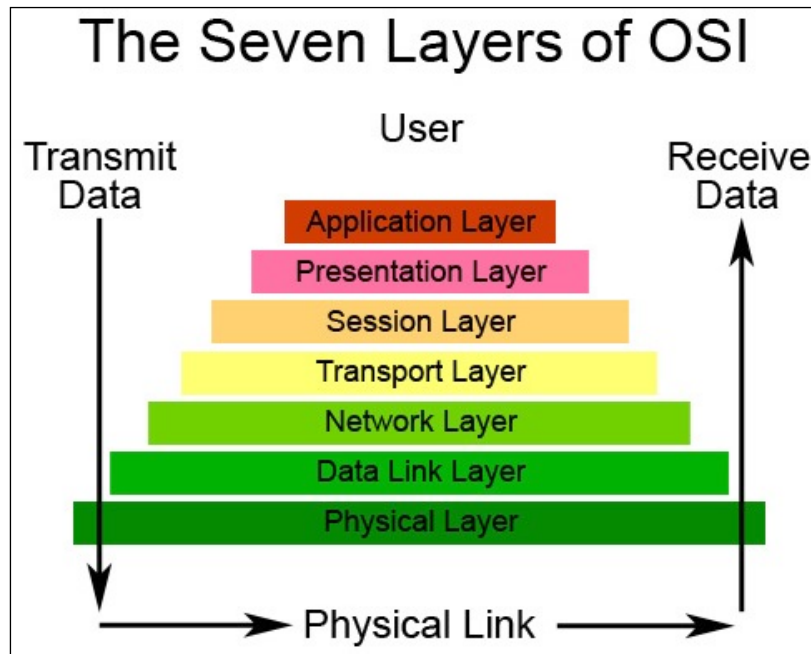
This chapter will look at two differing roles, that of the CTO / senior manager and the engineer, discussing at length some of the initiatives that can be utilized to facilitate the desired cultural changes that are required to create a successful DevOps transformation for a whole organization or even just allow a single department to improve their internal processes by automating everything they do.

In this chapter, the following topics will be covered:

- Initiating a change in behavior
- Top-down DevOps initiatives for networking teams
- Bottom-up DevOps initiatives for networking teams

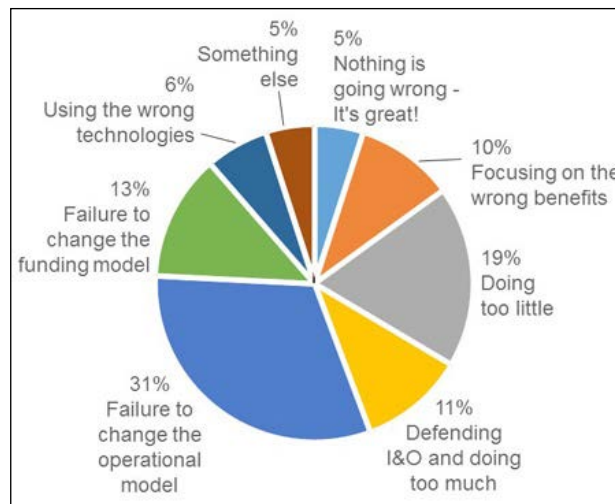
Initiating a change in behavior

The networking OSI model contains seven layers, but it is widely suggested that the OSI model has an additional eighth layer named the user layer, which governs how end users integrate and interact with the network. People are undoubtedly a harder beast to master and manage than technology, so there is *no one size fits all* solution to the vast amount of people issues that exist. The seven layers of OSI are shown in the following image:



Initiating cultural change and changes in behavior is the most difficult task an organization will face, and it won't occur overnight. To change behavior there must first be obvious business benefits. It is important to first outline the benefits that these cultural changes will bring to an organization, which will enable managers or change agents to make business justifications to implement the required changes.

Cultural change and dealing with people and processes is notoriously hard, so divorcing the tools and dealing with people and processes is paramount to the success of any DevOps initiative or project. Cultural change needs to be carefully planned and become a company initiative. In a recent study by Gartner, it was shown that selecting the wrong tooling was not the main reason that cloud projects were a failure, instead the top reason was failure to change the operational model:



Reasons to implement DevOps

When implementing DevOps, some myths are often perpetuated, such as DevOps only works for start-ups, it won't bring any value to a particular team, or that it is simply a buzz word and a fad.

The quantifiable benefits of DevOps initiatives are undeniable when done correctly. Some of these benefits include improvements to the following:

- The velocity of change
- Mean time to resolve
- Improved uptime
- Increased number of deployments
- Cross-skilling between teams
- The removal of the bus factor of one

Any team in the IT industry would benefit from these improvements, so really teams can't afford to not adopt DevOps, as it will undoubtedly improve their business functions.

By implementing a DevOps initiative, it promotes repeatability, measurement, and automation. Implementing automation naturally improves the velocity of change, increases the number of deployments a team can do in any given day and improves time to market. Automation of the deployment process allows teams to push fixes through to production quickly as well as allowing an organization to push new products and features to market.

A byproduct of automation is that the mean time to resolve will also become quicker for infrastructure issues. If infrastructure or network changes are automated, they can be applied much more efficiently than if they were carried out manually. Manual changes depend on the velocity of the engineer implementing the change rather than an automated script that can be measured more accurately.

Implementing DevOps also means measuring and monitoring efficiently too, so having effective monitoring is crucial on all parts of infrastructure and networking, as it means the pace in which root cause analysis can be carried out improves. Having effective monitoring helps to facilitate the process of mean time to resolve, so when a production issue occurs, the source of the issue can be found quicker than numerous engineers logging onto consoles and servers trying to debug issues.

Instead a well-implemented monitoring system can provide a quick notification to localize the source of the issue, silencing any resultant alarms that result from the initial root cause, allowing the issue to be highlighted and fixed efficiently.

The monitoring then hands over to the repeatable automation, which can then push out the localized fix to production. This process provides a highly accurate feedback loop, where processes will improve daily. If alerts are missed, they will ideally be built into the monitoring system over time as part of the incident post-mortem.

Effective monitoring and automation results in quicker mean time to resolve, which leads to happier customers, and results in improved uptime of products. Utilizing automation and effective monitoring also means that all members of a team have access to see how processes work and how fixes and new features are pushed out.

This will mean less of a reliance on key individuals removing the *bus factor* of one where a key engineer needs to do the majority of tasks in the team as he is the most highly skilled individual and has all of the system knowledge stored in his head.

Using a DevOps model means that the very highly skilled engineer can instead use their talents to help *cross skill* other team members and create effective monitoring that can help any team member carry out the root cause analysis they normally do manually. This builds the talented engineer's deep knowledge into the monitoring system, so the monitoring system as opposed to the talented engineer becomes the go-to point of reference when an issue first occurs, or ideally the monitoring system becomes the source of truth that alerts on events to prevent customer facing issues. To improve cross-skilling, the talented engineer should ideally help write automation too, so they are not the only member of the team that can carry out specific tasks.

Reasons to implement DevOps for networking

So how do some of those DevOps benefits apply to traditional networking teams? Some of the common complaints with siloed networking teams today are the following:

- Reactive
- Slow, often using ticketing systems to collaborate
- Manual processes carried out using admin terminals
- Lack of preproduction testing
- Manual mistakes leading to network outages
- Constantly in firefighting mode
- Lack of automation in daily processes

Network teams like infrastructure teams before them are often used to working in siloed teams, interacting with other teams in large organizations via ticketing systems or using suboptimal processes. This is not a streamlined or optimized way of working, it is scenarios such as this which led to the DevOps initiative being started, that sought to break down barriers between *Development* and *Operations* staff, but its remit has since widened.

Networking does not seem to have been initially included in this DevOps movement yet, but software delivery can only operate as fast as the slowest component. The slowest component will eventually become the bottleneck or blocker of the entire delivery process. That slowest component often becomes the star engineer in a siloed team that can't process enough tickets in a day manually to keep up with demand, thus becoming the bus factor of one. If that engineer takes a holiday or has a sick day, then work is blocked, the company becomes too reliant and cannot function efficiently without them.

If a team is not operating in the same way as the rest of the business, then all other departments will be slowed down as the siloed department is not agile enough. Put simply, the reason networking teams exist in most companies is to provide a service to development teams. Development teams require networking to be deployed, so they can test product changes and also deploy products to production, once deployed to production the business can start making money from those products.

Networking changes to ACL policies, load balancing rules, and provisioning of new subnets for new applications can no longer be deemed a success if they take days, months or even weeks. Networking has a direct impact on the *velocity of change, mean time to resolve, uptime*, as well as *the number of deployments*, which are four of the key performance indicators of a successful DevOps initiative. So networking needs to be included in a DevOps model by companies, otherwise all of these quantifiable benefits will become constrained.

Given the rapid way AWS, Microsoft Azure, OpenStack, and **Software-defined Networking (SDN)** can be used to provision network functions in the private and public cloud, it is no longer acceptable for network teams to not adapt their operational processes and learn new skills. But the caveat is that the evolution of networking has been quick, and they need the support and time to do this.

If a cloud solution is implemented and the operational model does not change, then no real quantifiable benefits will be felt by the organization. Cloud projects traditionally do not fail because of technology, cloud projects fail because of the incumbent operational models that hinder them from being a success. There is zero value to be had from building a brand new OpenStack private cloud, with its open set of extensible APIs to manage compute, networking, and storage if a company doesn't change its operational model and allow end users to use those APIs to self-service their requests.

If network engineers are still using the GUI to point and click and cut and paste then this doesn't bring any real business value as the network engineer that cuts and pastes the slowest is the bottleneck. The company may as well stick with their current technology and processes as implementing a private cloud solution with manual processes will not result in speeding up time to market or mean time to recover from failure.

However, cloud should not be used as an excuse to deride your internal network staff, as incumbent operational models in companies are typically not designed or set up by current staff, they are normally inherited. Moving to public cloud doesn't solve the problem of the operational agility of a company's network team, it is a quick fix and bandage that disguises the deeper rooted cultural challenges that exist.

However, smarter ways of working allied with use of automation, measurement, and monitoring can help network teams refine their internal processes and facilitate the developers and operations staff that they work with daily. Cultural change can be initiated in two different ways, grass roots bottom-up initiatives coming from engineers, or top-down management initiatives.

Top-down DevOps initiatives for networking teams

Top-down DevOps initiatives are when a CTO, director, or senior manager have to buy in from the company to make changes to the operational model. These changes are required as the incumbent operational model is deemed suboptimal and not set up to deliver software at the speed of competitors, which inherently delays new products or crucial fixes from being delivered to market.

When doing DevOps transformations from a top-down management level, it is imperative that some ground work is done with the teams involved, if large changes are going to be made to the operational model, it can often cause unrest or stress to staff on the ground.

When implementing operational changes, upper management need to have the buy in of the people on the ground as they will operate within that model daily. Having teams buy in is a very important aspect; otherwise, the company will end up with an unhappy workforce, which will mean the best staff will ultimately leave.

It is very important that upper management engage staff when implementing new operational processes and deal with any concerns transparently from the outset, as opposed to going for an offsite management meeting and coming back with an enforced plan, which is all too common theme.

Management should survey the teams to understand how they operate on a daily basis, what they like about the current processes and where their frustrations lie. The biggest impediment to changing an operational model is misunderstanding the current operational model. All initiatives should ideally be led and not enforced. So let's focus on some specific top-down initiatives that could be used to help.

Analyzing successful teams

One approach would be for the management to look at other teams within the organization whose processes are working well and are delivering in an incremental agile fashion, if no other team in the organization is working in this fashion, then reach out to other companies.

Ask if it would be possible to go and look at the way another company operates for a day. Most companies will happily use successful projects as reference cases to public audiences at conferences or meet-ups, as they enjoy showing their achievements, so it shouldn't be difficult to seek out companies that have overcome similar cultural challenges. It is good to attend some DevOps conferences and look at who is speaking, so approach the speakers and they will undoubtedly be happy to help.

Management teams should initially book a meeting with the high-performing team and do a question and answer session focusing on the following points, if it is an external vendor then an introduction phone call can suffice.

Some important questions to ask in the initial meeting are the following:

- Which processes normally work well?
- What tools do they actually use on a daily basis?
- How is work assigned?
- How do they track work?
- What is the team structure?
- How do other teams make requests to the team?
- How is work prioritized?
- How do they deal with interruptions?
- How are meetings structured?

It is important not to reinvent the wheel, if a team in the organization already has a proven template that works well, then that team could also be invaluable in helping facilitate cultural change within the networks team. It will be slightly more challenging if focus is put on an external team as the evangelist as it opens up excuses such as it being easier for them because of x, y, and z in their company.

A good strategy, when utilizing a local team in the organization as the evangelist, is to embed a network engineer in that team for a few weeks and have them observe and give feedback how the other teams operate and document their findings. This is imperative, so the network engineers on the ground understand the processes.

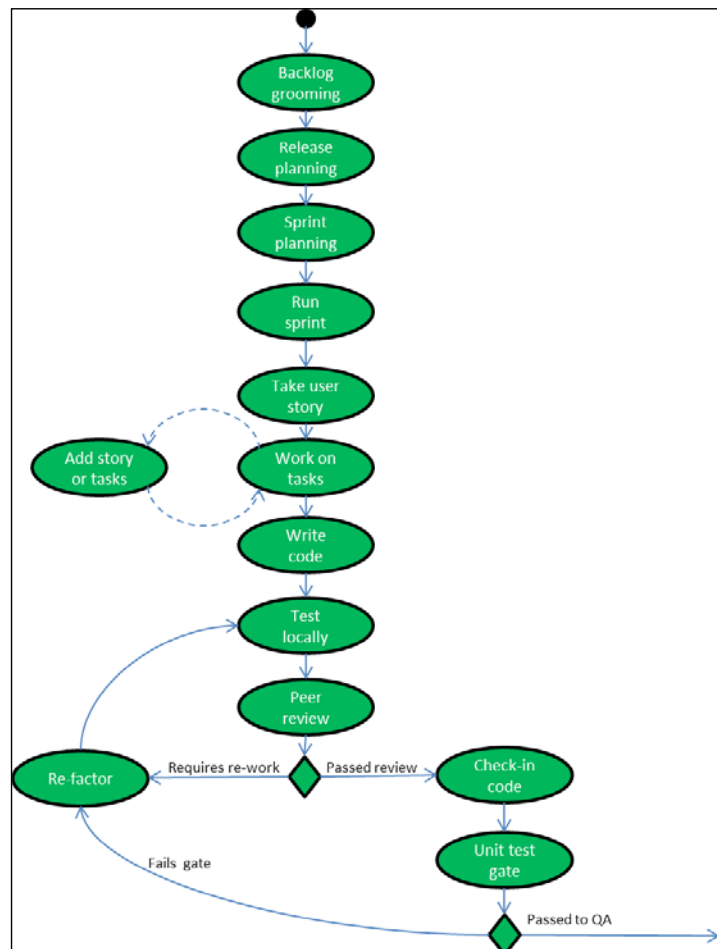
Flexibility is also important, as only some of the successful team's processes may be applicable to a network team, so don't expect two teams to work identically. The sum of parts and personal individuals in the team really do mean that every team is different, so focus on goals rather than the implementation of strict processes. If teams achieve the same outcomes in slightly different ways, then as long as work can be tracked and is visible to management, it shouldn't be an issue as long as it can be easily reported on.

Make sure pace is prioritized, select specific change agents to make sure teams are comfortable with new processes, so empower change agents in the network team to choose how they want to work by engaging with the team by creating new processes and also put them in charge of eventual tool selection. However, before selecting any tooling, it is important to start with process and agree on the new operational model to prevent tooling driving processes, this is a common mistake in IT.

Mapping out activity diagrams

A good piece of advice is to use an activity diagram as a visual aid to understand how a team's interactions work and where they can be improved.

A typical development activity diagram, with manual hand-off to a quality assurance team is shown here:



Utilizing activity diagrams as a visual aid is important as it highlights suboptimal business process flows. In the example, we see a development team's activity diagram. This process is suboptimal as it doesn't include the quality assurance team in the **Test locally** and **Peer review** phases. Instead it has a formalized QA hand-off phase, which is very late in the development cycle, and a suboptimal way of working as it promotes a development and QA silo, which is a DevOps anti-pattern.

A better approach would be to have QA engineers work on creating test tasks and creating automated tests, whereas the development team works on coding tasks. This would allow the development **Peer review** process to have a QA engineers' review and test developer code earlier in the development lifecycle and make sure that every piece of code written has appropriate test coverage before the code is checked in.

Another shortcoming in the process is that it does not cater for software bugs found by the quality assurance team or in production by customers, so mapping these streams of work into the activity diagram would also be useful to show all potential feedback loops.

If a feedback loop is missed in the overall activity diagram, then it can cause a breakdown in the process flow, so it is important to capture all permutations in the overarching flow that could occur before mapping tooling to facilitate the process.

Each team should look at ways of shortening interactions to aid mean time to resolve and improve the velocity of change at which work can flow through the overall process.

Management should dedicate some time in their schedule with the development, infrastructure, networking, and test teams and map out what they believe the team processes to be in their individual teams. Keep it high level, this should represent a simple activity swim-lane utilizing the start point where they accept work and the process the team goes through to deliver that work.

Once each team has mapped out the initial approach, they should focus on optimizing it and removing the parts of the process they dislike and discuss ways the process could be improved as a team. It may take many iterations before this is mapped out effectively, so don't rush this process, it should be used as a learning experience for each team.

The finalized activity diagram will normally include management and technical functions combined in an optimized way to show the overall process flow. Try not to bother using **Business Process Management (BPM)** software at this stage; a simple white board will suffice to keep it simple and informal.

It is a good practice to utilize two layers of an activity diagram, so the first layer can be a box that simply says **Peer review**, which then references a nested activity diagram outlining what the team's peer review process is. Both need refined but the nested tier of business processes should be dictated by the individual teams as these are specific to their needs, so it's important to give teams the flexibility they need at this level.

It is important to split the two tiers out; otherwise, the overall top layer of the activity diagram will be too complex to extract any real value from, so try and minimize the complexity at the top layer, as this will need to be integrated with other teams' processes. The activity doesn't need to contain team-specific details such as how an internal team's **Peer review** process operates as this will always be subjective to that team; this should be included but will be a nested layer activity that won't be shared.

Another team should be able to look at a team's top layer activity diagram and understand the process without explanation. It can sometimes be useful to first map out a high-performing teams' top layer activity diagram to show how an integrated joined-up business process should look.

This will help teams that struggle a bit more with these concepts and allow them to use that team's activity diagram as a guide. This can be used as a point of reference and show how these teams have solved their cross-team interaction issues and facilitated one or more teams interacting without friction. The main aim of this exercise is to join up business processes, so they are not siloed between teams, so the planning and execution of work is as integrated as possible for joined-up initiatives.

Once each team has completed their individual activity diagram and optimized it to the way the team wants, the second phase of the process can begin. This involves layering each team's top layer of their activity diagrams together to create a joined-up process.

Teams should use this layering exercise as an excuse to talk about suboptimal processes and how the overall business process should look end to end. Utilize this session to remove perceived bottlenecks between teams, completely ignoring existing tools and the constraints of current tools, this whole exercise should be focusing on process not tooling.

A good example of a suboptimum process flow that is constrained by tooling would be a stage on a top layer activity diagram that says raise ticket with ticketing system. This should be broken down so work is people focused, what does the person requesting the change actually require?

A Developers day job involves writing code and building great features and products, so if a new feature needs a network change, then networking should be treated as part of that feature change. So the time taken for the network changes needs to be catered for as part of the planning and estimation for that feature rather than a ticketed request that will hinder the velocity of change when it is done reactively as an afterthought.

This is normally a very successful exercise when engagement is good, it is good to utilize a senior engineer and manager from each team in the combined activity diagram layering exercise with more junior engineers involved in each team included in the team-specific activity diagram exercise.

Changing the network team's operational model

The network team's operational model at the end of the activity diagram exercise should ideally be fully integrated with the rest of the business. Once the new operational model has been agreed with all teams, it is time to implement it.

It is important to note that because the teams on the ground created the operational model and joined-up activity diagram, it should be signed off by all parties as the new business process. So this removes the issue of an enforced model from management as those using it have been involved in creating it. The operational model can be iterated and improved over time, but interactions shouldn't change greatly although new interaction points may be added that have been initially missed. A master copy of the business process can then be stored and updated, so anyone new joining the company knows exactly how to interact with other teams.

In the short term, it may seem the new approach is slowing down development estimates as automation is not in place for network functions, so estimation for developer features becomes higher when they require network changes.

This is often just a truer reflection of reality, as estimations didn't take into account network changes and then they became blockers as they were tickets, but once reported, it can be optimized and improved over time.

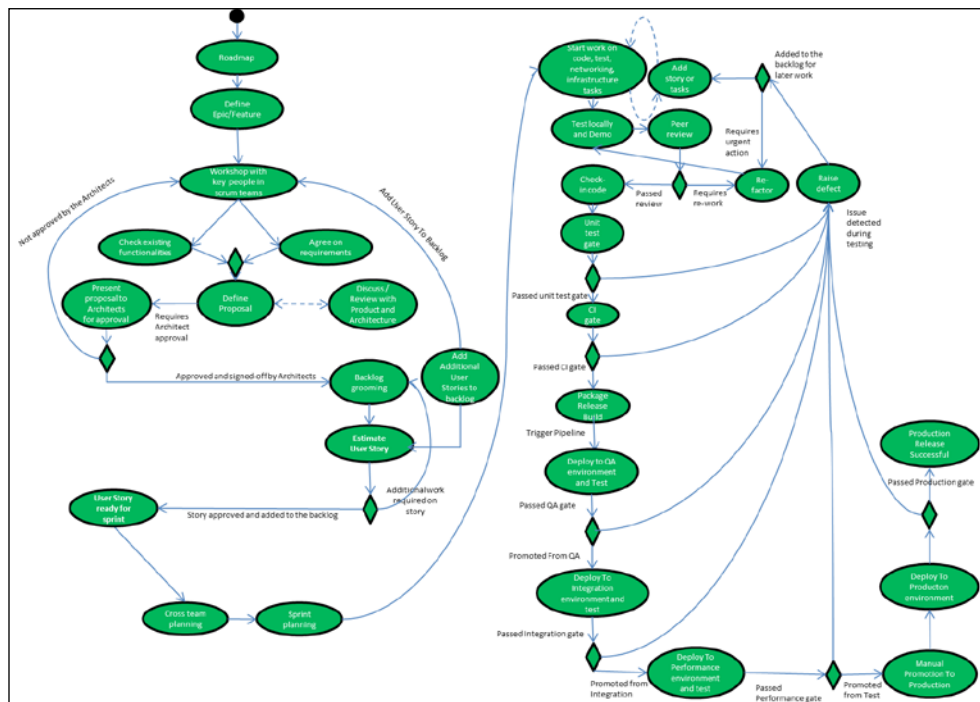
Once the overall activity diagram has been merged together and agreed with all the teams, it is important to remember if the processes are properly optimized, there should not be pages and pages of high-level operations on the diagram. If the interactions are too verbose, it will take any change hours and hours to traverse each of the steps on the activity diagram.

The activity diagram mentioned later shows a joined-up business process, where work is defined from a single roadmap producing user stories for all teams. New user stories, which are units of work, are then estimated out by cross-functional teams, including developers, infrastructure, quality assurance, and network engineers. Each team will review the user story and work out which cross-functional tasks are involved to deliver the feature.

The user story then becomes part of the sprint with the cross-functional teams working on the user story together making sure that it has everything it needs to work prior to the check-in. After **Peer review**, the feature or change is then handed off to the automated processes to deliver the code, infrastructure, and network changes to production.

The checked-in feature then flows through unit testing, quality assurance, integration, and performance testing quality gates, which will include any new tests that were written by the quality assurance team before check-in. Once every stage is passed, the automation is invoked by a button press to push the changes to production. Each environment has the same network changes applied, so network changes are made first on test environments before production.

This relies on treating networking as code, meaning automated network processes need to be created so the network team can be as agile as the developers.



Once the agreed operational model is mapped out only then should the DevOps transformation begin. This will involve selecting the best of breed tools at every stage to deliver the desired outcome with the focus on the following benefits:

- The velocity of change
- Mean time to resolve
- Improved uptime
- Increased number of deployments
- Cross-skilling between teams
- The removal of the bus factor of one

All business processes will be different for each company, so it is important to engage each department and have the buy-in from all managers to make this activity a success.

Changing the network team's behavior

Once a new operational model has been established in the business, it is important to help prevent the network team from becoming the bottleneck in a DevOps-focused Continuous Delivery model.

Traditionally, network engineers will be used to operating command lines and logging into admin consoles on network devices to make changes. Infrastructure engineers adjusted to automation as they already had scripting experience in **bash** and **PowerShell** coupled with a firm grounding in Linux or Windows operating systems, so transitioning to configuration management tooling was not a huge step.

However, it may be more difficult to persuade network engineers to make that same transition initially. Moving network engineers towards coding against APIs and adopting configuration management tools may initially appear daunting, as it is a higher barrier to entry, but having an experienced automation engineer on hand can help network engineers make this transition.

It is important to be patient, so try to change this behavior gradually by setting some automation initiatives for the network team in their objectives. This will encourage the correct behavior and try and incentivize it too. It may be useful to start off automation initiatives by offering training or purchasing particular coding books for teams.

It may also be useful to hold an initial automation hack day; this will give network engineers a day away from their day jobs and time to attempt to automate a small process, which is repeated everyday by network engineers. If possible, make this a mandatory exercise, so that the engineers have to participate and make other teams available to cover for the network team, so they aren't distracted. This is a good way of seeing which members of the network team may be open to evangelizing DevOps and automation. If any particular individual stands out, then work with them to help push automation initiatives forward to the rest of the team by making them the champion for process automation.

Establishing an internal DevOps meet-up where teams present back their automation achievements is also a good way of promoting automation in network teams and this keeps the momentum going. Encourage each team across the business to present back interesting things they have achieved each quarter and incentivize this too by allowing each team time off from their day job to attend if they participate. This leads to a sense of community and shows teams they are part of a bigger movement that is bringing real cost benefits to the business. This also helps to focus teams on the common goal of making the company better and breaks down barriers between teams in the process.

One approach that should be avoided at all costs is having other teams write all the network automation for the network team. Ideally, it should be the networking team that evolves and adopts automation, so giving the network team a sense of ownership over the network automation is very important. This though requires full buy-in from networking teams and discipline not to revert back to manual tasks at any point even if issues occur.

To ease the transition, offer to put an automation engineer into the network team from the infrastructure or developments teams, but this should only be a temporary measure. It is important to select an automation engineer that is respected by the network team and knowledgeable in networking, as no one should ever attempt to automate network processes that they cannot operate by hand, so having someone well-versed in networking to help with network automation is crucial, as they will be training the network team so have to be respected. If an automation engineer is assigned to the network team and isn't knowledgeable or respected, then the initiative will likely fail, so choose wisely.

It is important to accept at an early stage that this transition towards DevOps and automation may not be for everyone, so not every network engineer will be able to make the journey. It is all about the network team seizing the opportunity and showing initiative and willingness to pick up and learn new skills. Disruptive or negative behavior to new automation initiatives should be stamped out early on as it may have a bad influence on the network team.

It is fine to have for people to have a cynical skepticism at first, but not attempting to change or build new skills shouldn't be tolerated, as it will disrupt the team dynamic and this should be monitored so it doesn't cause automation initiatives to fail or stall, just because individuals are proving to be blockers or being disruptive.

Every organization has its own unique culture and a company's rate of change will be subject to cultural uptake of the new processes and ways of working. When initiating cultural change, change agents are necessary and can come from internal IT staff or external sources depending on the aptitude and appetite of the staff to change. Every change project is different, but it is important that it has the correct individuals involved to make it a success along with the correct management sponsorship and backing.

Bottom-up DevOps initiatives for networking teams

Bottom-up DevOps initiatives are when an engineer, team leads, or lower management don't necessarily have buy-in from the company to make changes to the operational model. However, they realize that although changes can't be made to the overall incumbent operational model, they can try and facilitate positive changes using DevOps philosophies within their team that can help the team perform better and make their productivity more efficient.

When implementing DevOps initiatives from a bottom-up initiative, it is much more difficult and challenging at times as some individuals or teams may not be willing to change the way they work and operate as they don't have to. But it is important not to become disheartened and do the best possible job for the business.

It is still possible to eventually convince upper management to implement a DevOps initiative using grass roots initiatives to prove the process brings real business benefits.

Evangelizing DevOps in the networking team

Try and stay positive at all times, working on a bottom-up initiative can be exhausting at times, but it is important to roll with the punches and not take things too personally. Always remain positive and try to focus on evangelizing the benefits associated with DevOps processes and positive behavior first within your own team. The first challenge is to convince your own team of the merits of adopting a DevOps approach before even attempting to convince other teams in the business.

A good way of doing this is by showing the benefits that the DevOps approach has made to other companies, such as Google, Facebook, and Etsy, focusing on what they have done in the networking space. A pushback from individuals may be the fact that these companies are unicorns and DevOps has only worked for companies for this reason, so be prepared to be challenged. Seek out initiatives that have been implemented by these companies that the networking team could adopt and are actually applicable to your company.

In order to facilitate an environment of change, work out what your colleagues' drivers are, what motivates them? Try tailoring the sell to individual's motivations, the sell to an engineer or manager may be completely different. An engineer on the ground may be motivated by the following:

- Doing more interesting work
- Developing skills and experience
- Helping automate menial daily tasks
- Learning sought-after configuration management skills
- Understanding the development lifecycle
- Learning to code

A manager on the other hand will probably be more motivated by offering to measure KPIs that make his team look better such as:

- Time taken to implement changes
- Mean time to resolve failures
- Improved uptime of the network

Another way to promote engagement is to invite your networking team to DevOps meet-ups arranged by forward-thinking networking vendors. They may be amazed that most networking and load balancing vendors are now actively promoting automation and DevOps and not yet be aware of this. Some of the new innovations in this space may be enough to change their opinions and make them interested in picking up some of the new approaches, so they can keep pace with the industry.

Seeking sponsorship from a respected manager or engineer

After making the network team aware of the DevOps initiatives, it is important to take this to the next stage. Seek out a respected manager or senior engineer in the networking team that may be open to trying out DevOps and automation. It is important to sell this person the dream, state how you are passionate about implementing some changes to help the team, and that you are keen to utilize some proven best practices that have worked well for other successful companies.

It is important to be humble, try not to rant or spew generalized DevOps jargon to your peers, which can be very off-putting. Always make reasonable arguments and justify them while avoiding to make sweeping statements or generalizations. Try not to appear to be trying to undermine the manager or senior engineer, instead ask for their help to achieve the goal by seeking their approval to back the initiative or idea. A charm offensive may be necessary at this stage to convince the manager or engineer that it's a good idea but gradually building up to the request can help otherwise it may appear insincere if the request comes out the blue. Potentially analyze the situation over lunch or drinks and gauge if it is something they would be interested in, there is little point trying to convince people that are stubborn as they probably will not budge unless the initiative comes from above.

Once you have found the courage to broach the subject, it is now time to put forward numerous suggestions on how the team could work differently with the help of a mediator that could take the form of a project manager. Ask for the opportunity to try this out on a small scale and offer to lead the initiative and ask for their support and backing. It is likely that the manager or senior engineer will be impressed at your initiative and allow you to run with the idea, but they may choose the initiative you implement. So, never suggest anything you can't achieve, you may only get one opportunity at this so it is important to make a good impression.

Try and focus on a small task to start with; that's typically a pain point, and attempt to automate it. Anyone can write an automation script, but try and make the automation process easy to use, find what the team likes in the current process, and try and incorporate aspects of it. For example, if they often see the output from a command line displayed in a particular way, write the automation script so that it still displays the same output, so the process is not completely alien to them.

Try not to hardcode values into scripts and extract them into a configuration files to make the automation more flexible, so it could potentially be used again in different ways. By showing engineers the flexibility of automation, it will encourage them to use it more, show others in the teams how you wrote the automation and ways they could adapt it to apply it to other activities. If this is done wisely, then automation will be adopted by enthusiastic members of the team, and you will gain enough momentum to impress the sponsor enough to take it forward onto more complex tasks.

Automating a complex problem with the networking team

The next stage of the process after building confidence by automating small repeatable tasks is to take on a more complex problem; this can be used to cement the use of automation within the networking team going forward.

This part of the process is about empowering others to take charge, and lead automation initiatives themselves in the future, so will be more time-consuming. It is imperative that the more difficult to work with engineers that may have been deliberately avoided while building out the initial automation are involved this time.

These engineers more than likely have not been involved in automation at all at this stage. This probably means the most certified person in the team and alpha of the team, nobody said it was going to be easy, but it will be worth it in the long run convincing the biggest skeptics of the merits of DevOps and automation. At this stage, automation within the network team should have enough credibility and momentum to broach the subject citing successful use cases.

It's easier to involve all difficult individuals in the process rather than presenting ideas back to them at the end of the process. Difficult senior engineers or managers are less likely to shoot down your ideas in front of your peers if they are involved in the creation of the process and have contributed in some way.

Try and be respectful, even if you do not agree with their viewpoints, but don't back down if you believe that you are correct, or give up. Make arguments fact based and non-emotive, write down pros and cons, and document any concerns without ignoring them, you have to be willing to compromise but not to the point of devaluing the solution.

There may actually be genuine risks involved that need addressed, so valid points should not be glossed over or ignored. Where possible seek backup from your sponsor if you are not sure on some of the points or feel individuals are being unreasonable.

When implementing the complex automation task work as a team, not as an individual, this is a learning experience for others as well as yourself. Try and teach the network team a configuration management tool, they may just be scared try out new things, so go with a gentle approach. Potentially stop at times to try out some online tutorials to familiarize everyone with the tool and try out various approaches to solve problems in the easiest way possible.

Try and show the network engineers how easy it is to use configuration management tools and the benefits. Don't use complicated configuration management tools as it may put them off. The majority of network engineers can't currently code, something that will potentially change in the coming years. As stated before, infrastructure engineers at least had a grounding in bash or PowerShell to help get started, so pick tooling that they like and give them options. Try not to enforce tools they are not comfortable with. When utilizing automation, one of the key concerns for network engineers is peer review as they have a natural distrust that the automation has worked. Try and build in gated processes to address these concerns, automation doesn't mean no peer review so create a lightweight process to help. Make the automation easy to review by utilizing source control to show diffs and educate the network engineers on how to do this.

Coding can be a scary prospect initially, so propose to do some team exercises each week on a coding or configuration management task. Work on it as a team. This makes it less threatening, and it is important to listen to feedback. If the consensus is that something isn't working well or isn't of benefit, then look at alternate ways to achieve the same goal that works for the whole team. Before releasing any new automated process, test it in the preproduction environment, alongside an experienced engineer and have them peer review it, and try to make it fail against numerous test cases. There is only one opportunity to make a first impression, with a new process, so make sure it is a successful one.

Try and set up knowledge-sharing sessions between the team to discuss the automation and make sure everyone knows how to do operations manually too, so they can easily debug any future issues or extend or amend the automation. Make sure that output and logging is clear to all users as they will all need to support the automation when it is used in production.

Summary

In this chapter, we covered practical initiatives, which when combined, will allow IT staff to implement successful DevOps models in their organization. Rather than just focusing on departmental issues, it has promoted using a set of practical strategies to change the day-to-day operational models that constrain teams. It also focuses on the need for network engineers to learn new skills and techniques in order to make the most of a new operational model and not become the bottleneck for delivery.

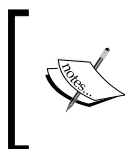
This chapter has provided practical real-world examples that could help senior managers and engineers to improve their own companies, emphasizing collaboration between teams and showing that networking departments is now required to automate all network operations to deliver at the pace expected by businesses.

Key takeaways from this chapter are that DevOps is not just about development and operations staff it can be applied to network teams. It is important to understand that before starting a DevOps initiative, take the time to analyze successful teams or companies and focus on what made them successful, but senior management sponsorship is the key to creating a successful DevOps model.

Your own company's model will not identically mirror other companies, so try not to copy like for like. Adapt the model so that it works in your own organization, allow teams to create their own processes, but don't dictate processes. Also, allow change agents to initiate changes that teams are comfortable with.

Try to automate all operational work, start small, and build up to larger, more complex problems once the team is comfortable with new ways of working. Always try and remember that successful change will not happen overnight. It will only work through a model of continuous improvement.

In the following chapters, we will look at ways of applying automation to networking and concentrate on configuration management tools such as **Ansible**. These configuration management tools can be used to increase the pace that network engineers can implement changes as well as making sure all network changes that are made are done in the same way and are less error-prone.



Useful links on DevOps are as follows:

<https://www.youtube.com/watch?v=TdAmAj3eaFI>

<https://www.youtube.com/watch?v=gqmuVHw-hQw>



4

Configuring Network Devices Using Ansible

This chapter will focus on some of the most popular networking vendors in the market today, namely Cisco, Juniper, and Arista, and look at how each of these market leading vendors have developed their own proprietary operating system to control network operations. The aim of this book is not to discuss which network vendor's solution is better, but instead look at ways network operators can utilize configuration management tooling today to manage network devices, now that most network vendors have created APIs and SDKs to programmatically control the network.

Once the basics of each operating system have been established, we will then shift focus to the hugely popular open source configuration management tool from Red Hat named Ansible (<https://www.ansible.com/>).

We will look at ways it can be used to configure network devices programmatically and assist with network operations. This chapter will show practical configuration management processes that can be used to manage network devices.

In this chapter, the following topics will be covered:

- Network vendors' operating systems
- Introduction to Ansible
- Ansible modules currently available for network automation
- Configuration management processes to manage network devices

Network vendors' operating systems

Market leading networking vendors, such as Cisco, Juniper, and Arista, have all developed their own operating systems that allow network operators to issue a series of commands to network devices via a **command-line interface (CLI)**.

Each vendor's CLI is run from their bespoke operating systems:

- Cisco Ios and Nxos
- Juniper Junos
- Arista Eos

All of these operating systems have meant that it has become easier to programmatically control switches, routers, and security devices provided by these vendors, as they seek to simplify operating network devices.

The rise of DevOps in industry has also meant that it is no longer acceptable to not provide programmatic APIs or SDK to aid automation, with networking vendors now integrating with configuration management tooling, such as Puppet, Chef, Ansible, and Salt, to plug into DevOps tool chains.

Cisco ios and Nxos operating system

The Cisco IOS operating system when released was the first of its kind, providing a set of command lines that network operators could use to mutate the state of the network. However, it still had its challenges; it had a monolithic architecture, which meant that all processes shared the same memory space, with no protection between parallel processes, so it didn't align itself well to parallel updates, but at the time it was the clear market leader. This changed network operations and meant that network engineers would each individually log onto network switches and routers to make updates using its fully featured CLI.

At the time, this greatly reduced the complexity of network operations, and Cisco standardized the way the networking industry carried out network operations in a data center. Network operators would log onto appliances and run an industry standard series of command lines to make changes to routers or switches, and Cisco ran certification programs to teach administrators how to operate the equipment and learn all the commands.

Today with efficiency and cost reductions key to businesses surviving and a shift towards more agile processes, this model in the modern data centers has an obvious scaling issue with x amount of network engineers required per network device.

The emergence of private clouds has meant that the number of network devices each network engineer needs to manage has grown dramatically, so automation has become key to managing the growing amount of devices in a consistent way. If a businesses competitors can put products to market quicker if they have automated operational models, then they will be able to put products to market quicker than organizations that are doing manual changes. Automation has become a necessity to keep up with the rapid churn of change required on the network. As IT is changing and evolving, then automation has become a prerequisite to facilitate that evolution.

Cisco, as the networking market has evolved in recent years, has since developed a new operating system named **Nxos**, which has allowed itself to integrate with open source technologies and lend itself to automation. The Nxos operating system is deployed with all new Nexus switches and routers, and this operating system has shifted Cisco towards open and modular standards by integrating with open protocols, such as **BGP**, **EVNP**, and **VXLAN**, and the appliances can even run **LXC** containers, which is an operating system-level virtualization method in order to run multiple isolated processes on a virtual machine or physical server.

Cisco have also provided a set of REST APIs that allows network operators to run native Linux and bash shells to carry out regular administration commands server side. In a world where AWS and OpenStack programmatic APIs are available to mutate network infrastructure, networking vendors needed to adapt to survive or they risked being left behind, so Cisco have made their own switches and routers as easy to configure and operate as the virtual appliances.

The Nxos operating system allows the use of the Red Hat enterprise Linux rpm package manager to control software updates. This means that software updates can be done on the Nxos in an industry standard way, the same as patching a Linux guest operating system would be carried out by an infrastructure system administrator. Consequently, Cisco network devices are now more intuitive to Linux system administrators and more like native Linux to end users, which has undoubtedly made them simpler to administrate.

The Cisco Nxos operating system means that the speed that network changes can be pushed increases, as operations staff can use their own tool chains and configuration management tools to automate updates. The Nxos operating system has become less vendor specific; therefore, lowering the barrier to entry to use networking products and automation of its product suites have become easier.

Juniper Junos operating system

The Juniper Junos operating systems driver is programmatically controlled to control network operations, Junipers Junos operating system was created to provide CLI that users can execute to retrieve facts about the running system. The Junos operating system is based on a clearly defined hierarchical model as opposed to using a series of unrelated configuration files. The hierarchical model also comes complete with operational and configuration modes of operation.

Intuitively, operational mode is used to upgrade the operating system, monitor the system, and also check the status of juniper devices. Configuration mode, on the other hand, allows network operators to configure user access and security, interfaces, hardware, and the set of protocols used on the device, which gives a clear separation of roles between those installing the system and those operating it. The Junos operating system supports all open protocols, such as BGP, VXLAN, and EVPN, as well as in-built roll forward and roll back capability.

Juniper provide a Python library named **PyEZ** for the Junos operating system as well as a PowerShell option for Windows administrators that utilizes PowerShell wrapped in Python. The Python library PyEZ can retrieve any configuration information using tables and views that allow network operators to script against runtime information provided by the Junos operating system. Once a table items have been extracted by utilizing a python script using a `get ()` method, tables can subsequently be treated as a Python dictionary and iterated, which allows users to carry out complex scripting if required, allowing network operators to automate all network operations. The Junos PYEZ library is also fully extensible and network operators can add functionality they deem appropriate using its widget system.

Arista EOS operating system

The Arista EOS operating system is based on open standards to promote automation of network functions. It relies upon a centralized **CloudVision eXchange (CVX)** and the CVX servers hold the centralized state of the network. The EOS operating system separates the functional control on every switch using **Sysdb**, which is the Arista EOS operating systems database. The Arista Sysdb is an in-memory database running in user space and contains the complete state of the Arista switch. Sysdb is maintained in memory on the device so if an Arista switch is either restarted or powered down all information for that switch is lost.

The CVX server acts as an aggregator managing all the state information from every switch's Sysdb into a network-wide database depending on what services are enabled on the cluster of CVX servers. When state changes occur to Sysdb on a switch then the change is pushed to the CVX centralized database, which then updates its configuration and notifies agents running on CVX of the change.

The Arista EOS operating system supports modern open protocols, such as MLAG, ECMP, BGP, and VXLAN. It utilizes overlay technologies such as VXLAN allowing applications to be deployed and remain portable in the modern data center. Arista heavily promotes the use of the Leaf-Spine architecture with ECMP, which allows a scale out model to be implemented; this aligns itself to modern cloud solutions such as OpenStack and makes it agnostic to SDN controller solutions.

The Arista EOS operating system is a Linux-based operating system designed to be programmatically controlled. The main driver for the EOS operating system is to allow network operators to carry out network operations' using a well structured set of APIs including the eAPI, CLI command as well as Python, Ruby, and GO libraries available as part of its SDK portfolio.

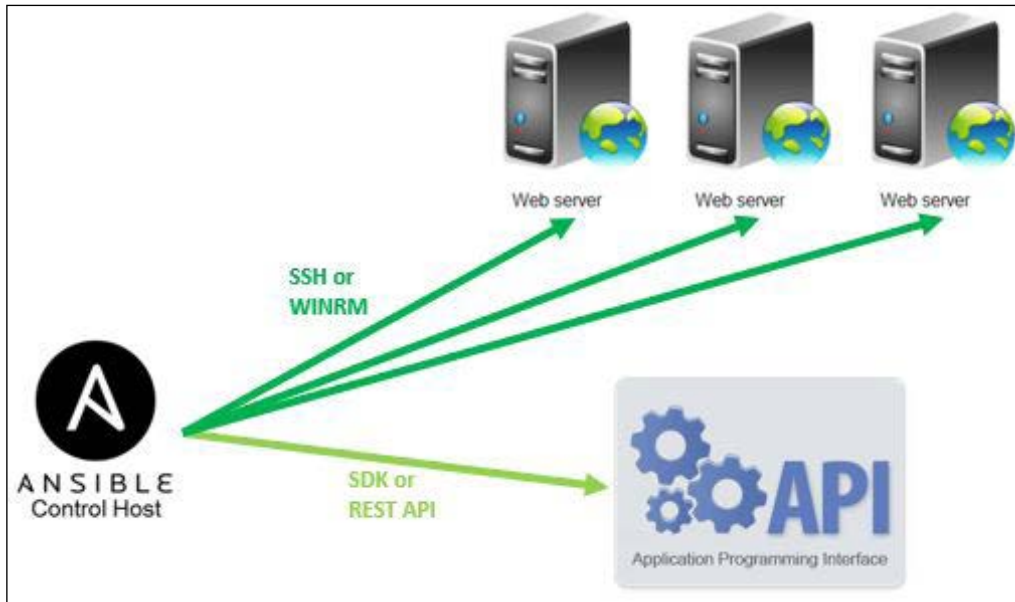
The EOS operating system also allows **Smart System Upgrade (SSU)** to allow scale out of Arista appliances with live patching and upgrades simplified and made more intuitive, this helps to support businesses 99.99% uptime targets. Switches can now be racked and cabled in the data center by data center operations teams, then handed over to Arista's **Zero Touch Provisioning (ZTP)** process that automates the initialization of switches and **Zero Touch Replacement (ZTR)** allows switches to be replaced in the data center.

The Arista EOS solution CVX product can be used to automate networking workflow tasks through the portal if users require a visual view of switches and routers and the CVX allows integration with SDN controllers using OVSDB, eAPI, or OpenFlow. Like Cisco and Juniper, the EOS API lends due to it having multiple SDK options so Arista products can be easily managed by configuration management tools, such as Puppet, Chef, Ansible, and Salt, so that no network operation needs to be carried out manually.

Introduction to Ansible

Ansible is primarily a push-based configuration management tool that uses a single **Ansible Control Host**, and it can connect to multiple Linux guest operating systems via SSH to configure them and recently added WinRM support, so it can now also configure Windows guests in the same way as Linux-based operating systems. As Ansible can connect to multiple servers simultaneously, it aids operators by allowing them to carry out uniform operations across multiple Linux or Windows servers at the same time. This allows Ansible to help simplify the automation of repeatable tasks by defining them in YAML, so they can be consistently executed against target servers. Ansible can also be used as a centralized orchestration tool that can connect to API endpoints and sequence API operations.

Here, we can see an example of the way an Ansible Control Host connects to servers or acts as a centralized orchestration tool:



Every operation that Ansible carries out should be idempotent as a standard, meaning that if the desired state is already configured on a server, then Ansible will check the intended state from a playbook or role and not take any action if a server is already in the correct state. Only if the state is different from what is specified in a playbook or role will the operation be executed to mutate the state of the server.

Ansible is a Python-based configuration management tool that controls servers from a Linux-based Control Host, using YAML files to define and describe desired state. Ansible is packaged with a rich set of extensible modules, which are primarily written in Python, but can also be written in any language that a user wishes. Ansible modules allow Python SDKs or REST API's to be wrapped in Ansible's plug-in boilerplate and then utilized from Ansible roles or playbooks in an easy-to-use architecture. Before going into more detailed examples, it is important to understand some of the Ansible terminology.

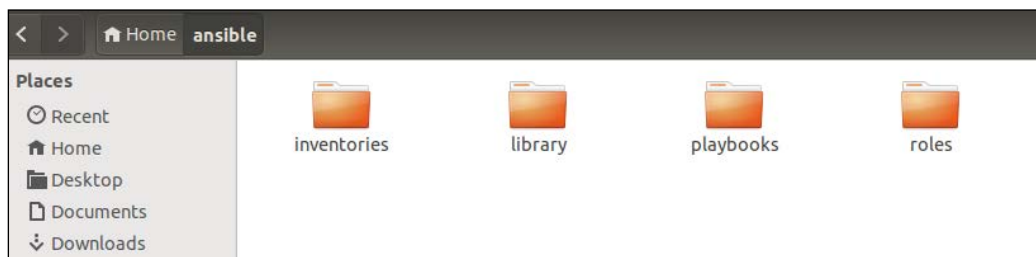
Ansible directory structure

Ansible is made up of a series of YAML files that are laid out in a customizable directory structure.

In this customized structure, the Ansible Controller Node has the following directory structure:

- The `inventories` folder holds the Ansible inventory
- The `library` folder holds any custom python plugins
- The `playbooks` folder holds all playbooks
- The `roles` folder holds all the Ansible roles

The overall directory structure is shown here:



This provides logical groupings of all Ansible components, which will be useful as the amount of playbooks or roles grow in size. It is best practice to version the `ansible` folder structure in a source control management system such as **Git**. Git is a distributed open source version control repository, which is designed to version control development code to facilitate speed and efficiency (<https://en.wikipedia.org/wiki/Git>).

Ansible inventory

An Ansible inventory file is simply a set of DNS hostnames or IP addresses defined in a YAML file. This allows Ansible to connect to those target hosts and execute specific commands on servers.

Ansible allows users to use inventory files to group servers into particular types or use cases. For example, in networking terms, when utilizing Ansible to set up a Leaf-Spine architecture, a network operator could have a group for Leaf switches and another for the Spine switches. This is because a different set of run-book commands would be required to configure each, so limits can be applied upon execution to only execute a command against a small subset of servers limited to one particular group.

An example of an inventory file defining Leaf and Spine switches can be found in the following image, showing the definition of two groups in the inventory file, one for Leaf switches named `leaf` and one for Spine switches named `spine` containing all the DNS entries for the switches:

```
[spine]
spineswitch01
spineswitch02

[leaf]
leafswitch01
leafswitch02
leafswitch03
leafswitch04
```

The same inventory can be described in an abbreviated format:

```
[spine]
spineswitch[01-02]

[leaf]
leafswitch[01-04]
```

Ansible modules

An Ansible module is typically written in Python or can be written in any other programming language. An Ansible module's code defines a set of operations to add or remove functionality from a guest operating system or alternately execute a command against an API if it is being used for orchestration. Ansible modules can be used to wrap either a simple command line, API call or any other operation a user desires that can be coded programmatically. Modules are set up, so they can be reused in multiple playbooks or roles in order to promote reusing code and the standardization of operations.

Code specified in an Ansible module is wrapped in Ansible's module boilerplate, which structures the layout of the module. The boilerplate promotes a set of standards, so each module is idempotent by design, meaning that the code will first detect the state of the system and then determine if a change in state is required or not before executing the operation.

When a state change is executed in Ansible, it is denoted by a yellow output on the console. If no action is taken, it will display the color green to state that the operation ran successfully, but no state change was made, whereas a red console output indicates a failure on the module.

Ansible modules expose a set of command-line arguments for the module that can either be mandatory or optional and can have default values. Modules that adhere to the Ansible standard are created with a state variable that contains `present` or `absent`, as one of the command-line variables. A module, when set to `present`, will add the feature that has been specified by the playbook and when it is set to `absent`, it will remove the specified feature. All modules will typically have code to deal with both of these use cases.

Once an Ansible module has been written, it is placed in the `library` folder, which means that it is available as a library to the Python interpreter and the code can then be utilized by defining it in an Ansible `playbooks` or `roles`. Ansible comes with a set of prepackaged core and extras modules that can all be accessed by writing some YAML to describe the operation that is required, all modules are packaged with documentation that are part of the boilerplate and available on the Ansible website.

Core modules are maintained by the Ansible core team in joint initiatives with software vendors and are generally of high quality. Extras modules can also be of a good quality but are not maintained by vendors and sometimes maintained by users that have committed back the modules to Ansible to help out the open source community.

A simple core `yum` module donated by `yum`: can be seen in the following screenshot that takes two command-line variables `name` which is used to specify the rpm to install and `state`, which determines whether to install or remove it from the target server:

```
- name: install the latest version of Apache
  yum: name=httpd state=present
```

Ansible roles

Roles are a further level of abstraction in Ansible and also defined using YAML files. Roles can be called from playbooks; this aims to simplify playbooks as much as possible. As increased sets of functionality are added to `playbooks`, they can become cluttered and difficult to maintain from a single file. So roles allow operators to create minimal playbooks that then pull all the information from the Ansible directory structure, which then determines the configuration steps that need to execute on servers or be run locally.

Ansible roles attempt to strip out repeatable parts of playbooks and group them, so they can be used by multiple playbooks if required. Roles are groupings to determine what the server profile should actually be, rather than just focusing on multiple ad hoc instructions, so a playbook could be named `spine.yml` and the playbook could contain a set of modular roles used to define the particular Spine switches run-list, when executed this playbook will build the Spine switch on each target server specified in the Ansible inventory. If designed correctly some of these roles should be modular enough that they can be reused when creating Leaf switches.

Ansible playbooks

An Ansible playbook is a YAML file that dictates the run-list to carry out on a particular set of host servers that are defined in an inventory file. A playbook specifies an ordered set of instructions to execute commands locally from Ansible Controller Node or on a target set of hosts specified in the Ansible inventory file.

An Ansible playbook can be used to create a run-list that calls out to modules or specific roles, which dictate the operations that should be executed against a server.

In this example, we see a playbook targeting the `spine` hosts in the inventory file and executing multiple `roles` to set up the Spine servers:

```
---
- hosts: spine
  gather_facts: no
  connection: local

  roles:
    - common
    - interfaces
    - bridging
    - ipv4
    - bgp
```

An alternate playbook could not use roles at all and call Ansible `yum` core module directly to install the `apache httpd-2.2.29` yum package on the inventory group named `server`:

```
---
- hosts: server
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
    yum: name=httpd-2.2.29 state=present
```

Playbooks can also specify when conditions to dictate if an action in the playbook should be executed or not based on the output of a proceeding operation. The `register` command is used to store JSON output from a task that can then be utilized in playbooks or roles by subsequent tasks to validate if they should be invoked by reading the result of the JSON and evaluating the `when` condition.

Ansible playbooks from version 2.x onwards can now utilize block rescue functionality too. So if an operation nested in a block command fails, then the rescue section of the playbook is invoked. This can be useful for doing cleanup of failed actions to make playbooks more robust.

The usefulness of a block rescue operation shouldn't be underestimated, when requiring to copy a large database `dmp` file to a backup location this operation could sometimes be error-prone due to the volume of data being copied. So if the disk space is too low on the target directory, then that operation could fail half way through leaving only part of the file copied and the server in an unusable state and the server could run out of disk space. Therefore, a rescue command could be used to clean up the copied file immediately, so the server isn't left in a bad state if the copy operation fails. After the rescue command has completed, the playbook will exit with an error but remain in its original state.

In the following example, we can see a playbook using the `copy:` module to copy the source file `/var/files/db.dmp` to `/backups/db.dmp` and the `file:` module being used to delete the file if the original command fails:

```
---
- hosts: servers
  remote_user: root
  tasks:
  - block:
    - copy: src=/var/files/db.dmp dest=/backups/db.dmp owner=armstrongs group=admin mode=0644
  rescue:
  - file: path=/backups/db.dmp owner=armstrongs state=absent group=admin mode=0644
```


Executing an Ansible playbook

After playbook and inventory files have been created utilizing the specified folder structure, it can now be executed by specifying the `ansible-playbook` command.

In the following example, the:

- `ansible-playbook` tells Ansible that a YAML playbook file should be specified
- `-i` flag is used to specify the inventory file
- `-l` limits the execution only to the servers under the inventory group (servers)
- `-e` passes additional variables to the playbook in this example production
- `-v` sets the verbosity of the output:

```
ansible-playbook -i inventories/inventory -l servers -e
environment=production playbooks/devops-for-networking.yml -v
```

Ansible var files and jinja2 templates

Ansible `var` files are just another YAML file that specify a set of variables that will be substituted into a playbook at runtime using the Ansible `include_vars` statement.

The `var` files are just a way of breaking out variables that are required by playbooks or roles at runtime. This means that different `var` files can be passed at runtime without having to hardcode variables into playbooks or roles.

An example of a `var` file syntax is shown in the following screenshot, this shows the contents of a `common.yml` `var` file containing one defined variable named `cert_name`:

```
# sslcert vars
cert_name: cert1
```

The following example shows the `common.yml` variable above and other `environment.yml` variables, both being loaded into the playbook. The `{{ environment }}` is useful as it means that different values could be passed from the `ansible-playbook` command line to control the variables that are imported into the playbook using the `-e "environment=production"` option at runtime:

```
- name: Include vars
  include_vars: "../roles/networking/vars/{{ item }}.yaml"
  with_items:
    - "common"
    - "{{ environment }}"
```

The `common.yaml` var files variables value `cert1` can then be used by specifying `{{ cert_name }}` variable in the playbook:

```
"{{ cert_name }}"
```

Ansible also has the ability to utilize Python jinja2 templates that can be transformed at runtime, to populate the configuration files information utilizing a set of var files; for example, the `{{ environment }}` variable in the preceding example can be specified at runtime to load variables that populate unique environment information. The jinja2 template once transformed using the `template` module will be parameterized to use the variables specified in the `environment.yaml` file.

In the following example, we can see the Ansible `template:` module being executed as part of a role copying a jinja2 template `network_template.j2` and transforming it to `/etc/network.conf`:

```
- template: src=/networking/network_template.j2 dest=/etc/network.conf owner=bin group=admin mode=0644
```

Prerequisites using Ansible to configure network devices

The base constructs covered in the *Introduction to Ansible* section in this chapter are all relevant to the Ansible networking modules, and to a networking team wishing to utilize Ansible for configuration management. Before starting, it is important to check with the networking vendors that the version of the networking operating system can be used with Ansible. The next step is to configure a small provisioning server to utilize as the Ansible Control Host, this is typically created on the management network so it has access appropriate to all switches.

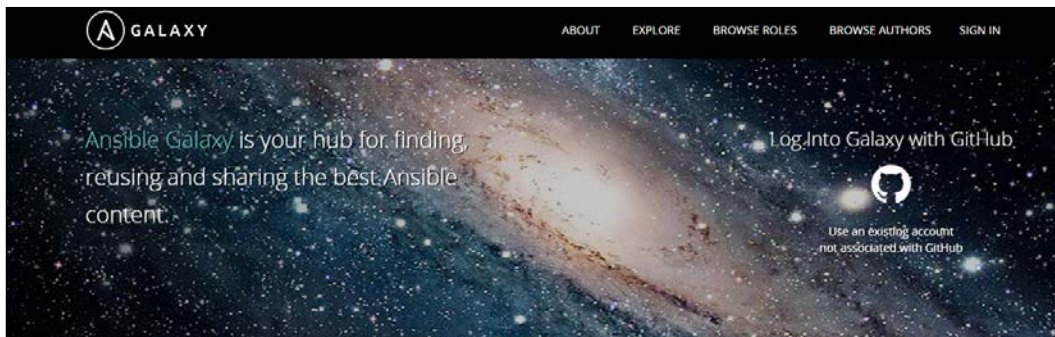
The provisioning server can be relatively small in size as it will just be required to connect over SSH to the Linux-based networking operating systems. Ensure that the API command line is enabled on the network device. It is also a good idea to create a temporary user account on each of the networking devices, which will allow you to set up a public key on the Ansible Control Host and **Secure Copy (SCP)** the created `id_rsa.pub` to the `authorized_keys` folder on the network devices using the temporary account. This will allow Ansible to use that private key to connect to all of the hosts without the need for dealing with passwords. The temporary password can then be deleted from each of the network devices once this setup activity has been completed, you could even use Ansible to do this as a first activity.

All being well, the next step would be to create the Ansible folder structure on the provisioning server and fill out the Ansible inventory file with all the DNS names of all the network devices and finally install Ansible when you are ready to start executing playbooks. Ansible is now packaged by Red Hat in rpm format, so this should just be a simple yum install as long as the Ansible Control Host has outbound Internet access to the Red Hat repositories when using a centos image or Red Hat Enterprise Linux. Ansible will of course work on any Linux-based operating system as is still available as a PyPi package that can be installed on Ubuntu.

Ansible Galaxy

If a network operator is looking for a start point and not well-versed in coding, they could look for examples on Ansible Galaxy, which hosts open source community roles that carry out many complex commands.

The network engineer can navigate to the Ansible Galaxy repository at <https://galaxy.ansible.com/>.



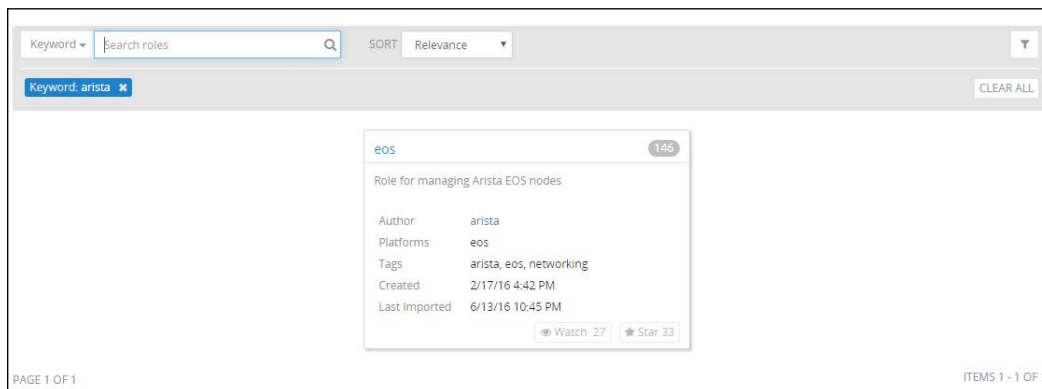
Ansible Galaxy houses thousands of Ansible roles that have been developed by the Open Source community.

Some available examples of networking roles are the Arista EOS role that can be used to automate Arista switch devices. Alternately, the Cisco EVPN VXLAN Spine role can be used to build Spine switches on Cisco devices or the Juniper Junos role can be used to automate Juniper network devices. So there is a wide variety of modules for a variety of technologies and use cases.

Take a look at the following useful links:

- Arista EOS (<https://galaxy.ansible.com/arista/eos-system/>)
- Cisco (https://galaxy.ansible.com/rogerscuall/evpn_vxlan-spine/)
- Juniper (<https://galaxy.ansible.com/Juniper/junos/>)

Users can browse roles and search for a particular networking vendor. In this example, a search for Arista has returned the **eos** role, as shown in the following screenshot:



Each role returned has a link to their corresponding GitHub repository:

The screenshot shows the Ansible Galaxy page for the role 'arista.eos'. The page title is 'arista.eos' and the description is 'Role for managing Arista EOS nodes'. There are two tabs: 'Details' (selected) and 'README'. Below the tabs, there is a 'Downloads' badge showing 146 downloads. There are also buttons for 'Issue Tracker', 'Github Repo', 'Watch 27', and 'Star 33'. The 'Minimum Ansible Version' is 1.9. The 'Installation' section shows the command '\$ ansible-galaxy install arista.eos'. The 'Tags' section shows 'arista', 'eos', and 'networking'. The 'Created' date is 02/17/2016 16:42:04 PM and the 'Imported' date is 06/13/2016 22:45:10 PM. The 'Version History' section shows a table with two columns: 'Version' and 'Release Date'. The table has one row: 'v1.3.0' and '02/17/2016 21:29:09 PM'.

Version	Release Date
v1.3.0	02/17/2016 21:29:09 PM

Ansible Galaxy is a very useful tool, where users can take roles as a start point and customize them to meet their needs. Rather than just taking from the community, any new roles that may be of use to others should be contributed back to the Ansible community.

Ansible core modules available for network operations

Since the release of Ansible 2.0, the Ansible configuration management tool been packaged with some of the core networking modules from Arista, Citrix, Cumulus, and Juniper. Ansible can be used to edit configuration for any network device. It isn't restricted to just these modules. Ansible Galaxy has a wide range of roles that have been developed by the open source community.

A subset of the Ansible 2.x networking modules can be shown in the following screenshot focusing upon the Juniper **Junos**, Arista **Eos**, Cisco **Nxos**, and **Ios**:

Junos	Nxos
<ul style="list-style-type: none"> • junos_command - Execute arbitrary commands on a remote device running Junos • junos_config - Manage configuration on remote devices running Junos • junos_facts - Collect facts from remote device running Junos • junos_netconf - Configures the Junos Netconf system service • junos_package - Installs packages on remote devices running Junos • junos_template - Manage configuration on remote devices running Junos 	<ul style="list-style-type: none"> • nxos_command - Run arbitrary command on Cisco NXOS devices • nxos_config - Manage Cisco NXOS configuration sections • nxos_facts - Gets facts about NX-OS switches • nxos_feature - Manage features in NX-OS switches • nxos_interface - Manages physical attributes of interfaces • nxos_ip_interface - Manages L3 attributes for IPv4 and IPv6 interfaces • nxos_nxapi - Manage NXAPI configuration on an NXOS device. • nxos_ping - Tests reachability using ping from Nexus switch • nxos_switchport - Manages Layer 2 switchport interfaces • nxos_template - Manage Cisco NXOS device configurations • nxos_vlan - Manages VLAN resources and attributes • nxos_vrf - Manages global VRF configuration • nxos_vrf_interface - Manages interface specific VRF configuration • nxos_vrrp - Manages VRRP configuration on NX-OS switches
<p>Eos</p> <ul style="list-style-type: none"> • eos_command - Run arbitrary command on EOS device • eos_config - Manage Arista EOS configuration sections • eos_eapi - Manage and configure EAPI. Requires EOS v4.12 or greater. • eos_template - Manage Arista EOS device configurations 	
	<p>Ios</p> <ul style="list-style-type: none"> • ios_command - Run arbitrary commands on ios devices. • ios_config - Manage Cisco IOS configuration sections • ios_template - Manage Cisco IOS device configurations over SSH

Ansible 2.x has sought to simplify networking modules by giving them a standard set of operations across all modules to make it feel more intuitive to network engineers. As many network engineers are not familiar with configuration management tooling, having a set of standards across modules simplifies the initial barrier to entry. As network engineers are able to see commands that they would utilize everyday being used as part of a playbook or a role, so Ansible can initially be utilized as a scheduling tool, before network operators delve into more complex modules.

One of the main fears network engineers have when first using configuration management tooling is not trusting the system or understanding what is going on under the covers. So, being able to easily read playbooks or roles and see the operations that are being executed builds confidence in the tooling and makes adoption easier.

It is fully expected that more complex networking modules will be built out over time by the open source community some of which are already available with roles from Arista, Juniper, and Cisco available in Ansible Galaxy. However, the following Ansible core modules have been standardized to allow configuration of Arista, Cisco, and Juniper network devices in the same way. These modules can be used in any playbook or role.

The `_command` module

The main module packaged with a vendor's networking modules in Ansible 2.x is the `_command` module. This is a conscious choice by Ansible as it is more intuitive to network engineers initially to use native network commands when switching to configuration management tooling.

This module allows Ansible to connect to hosts using SSH as network device's operating systems are primarily Linux-based operating systems.

The `_command` module allows network operators to apply configuration changes to switches by connecting from the Ansible Control Host. The syntax used by Ansible on this command is identical to what network operators would execute on network devices using CLI.

In the following example, the EOS command `show ip bgp summary` is executed by the `eos_command`, and it connects to every specified `{{ inventory_hostname }}`, which is a special Ansible variable that substitutes the DNS name of every node listed in the host group specified in the inventory file. It then registers the output of the command in the `eos_command_output` variable.

```
tasks:
  - name: execute show ip bgp
    eos_command:
      commands:
        - show ip bgp summary
      host={{ inventory_hostname }}
    register:
      eos_command_output
```

Junos syntax is identical. In the following example, a similar network command executed on Junos to show interfaces with the JSON output captures in the `junos_command_output` variable.

```
tasks:
  - name: show interfaces and capture in variable
    junos_command:
      commands:
        - show interfaces
    register:
      junos_command_output
```

The Cisco example shows Nxos, but the configuration is also the same in IOS. The `nxos_command` command issues a `show version` command and places the result in the `nxos_command_output` variable:

```
tasks:
  - name: show version and capture in variable
    nxos_command:
      commands:
        - show version
    register:
      nxos_command_output
```

The `_config` module

The `_config` module is used to configure updates in a deterministic way that could be used to implement change requests, by batching up a number of commands.

This module allows operators to update selected lines or blocks of running configuration programmatically on the network device. The module will connect to the device, extracting the running configuration before pushing batch updates in a completely deterministic way.

In the following example, the Arista switches configuration will be loaded by the module. The `no spanning-tree vlan 4094` command will be executed on the EOS operating system if the running configuration doesn't match the existing state, so the desired end state will be implemented on the switch.

```
tasks:
  - name: set no spanning tree on vlan
    eos_config:
      lines:
        - no spanning-tree vlan 4094
      host={{ inventory_hostname }}
    register:
      eos_command_output
```


The `_template` module

The `_template` module is used to update configuration utilizing a jinja2 template file. This can be extracted from the running configuration of a network device, updated and then pushed back to the device.

Another use case for the `_template` module would be allowing network administrators to extract the running config into a jinja2 template from one network device and apply it to other's switches to propagate the same changes.

The `_template` module will only push incremental changes unless the `force` command is specified as a command-line variable, which will carry out overwrite.

In the following example, the `eos_config` jinja2 template is pushed to the Arista device and will do an incremental change to the configuration if the jinja2 template has configuration changes.

```
tasks:
  - name: push eos_config.j2 template to EOS
    eos_template:
      src: eos_config.j2
    register:
      eos_command_output
```

Configuration management processes to manage network devices

DevOps is primarily all about people and process, so just focusing on some examples of playbooks or roles in isolation against a switch or firewall wouldn't help network engineer deal with the real-world networking challenges that they encounter every day. Selecting the correct tooling to facilitate processes is also important after the actual goals of a project have been established. Tooling should be selected after the business requirements have been made clear and not the opposite way round.

A network engineer could easily type in those commands into a network operating system as they could type commands into an Ansible playbook, so it is important to look at where the use of a configuration management tool such as Ansible adds real business value.

Implementing a new tool in isolation doesn't really help the network teams improve efficiency as a standalone activity, but the modules that have been created in Ansible to manage Arista, Juniper, and Cisco are facilitators of process that help simplify and standardize processes and approaches. However, it really is the process that wraps and utilizes these modules that is the key differentiator.

Ansible can be used to help with network operations in many ways, but it is good to try and categorize tasks into the following categories:

- Desired state
- Change requests
- Self-service operations

Desired state

A day one set of playbooks should be used to set the desired state of the network, utilizing a set of roles and modules to build out brand new network devices and are and control the network's intended state. An example of a day one playbook could be the first time a network engineer needs to configure a Leaf-Spine architecture utilizing Arista Leaf and Spine switches, which can seem a pretty daunting activity at first. But the beauty is that the state of the whole underlay network could be described in Ansible, but the same can be said for a firewall or any other device.

In the case of the Leaf-Spine network, activities will include configuring multiple Leaf and Spine switches, so creating a set of roles to abstract the common operations and calling them from a playbook is desirable, as the same configuration will need to be carried out on multiple servers.

A network engineer will begin by setting up the Ansible Control Host as covered in the Ansible prerequisites section. They will then create their inventory file for the Leaf-Spine architecture to configure the network devices.

The network engineer should define the inventory for all the network devices they plan to configure. In the following example, we see two host groups containing two spine switches and four leaf switches:

```
[spine]
spineswitch[01-02]

[leaf]
leafswitch[01-04]
```

The network operator will also need to specify the playbook containing the roles that they wish to execute in the `spine.yml` playbook, as shown in the following screenshot, to first build out the Spine switches with the desired configuration.

In the following example playbook, we see that the playbook targets the Spine host group and executes `common`, `interfaces`, `bridging`, `ipv4`, and `bgp` roles against the servers:

```
---
- hosts: spine
  gather_facts: no
  connection: local

  roles:
    - common
    - interfaces
    - bridging
    - ipv4
    - bgp
```

The executed roles carry out the following configuration:

- **common role:** This role is used to configure the IP routing table on the Spine
- **interfaces role:** This role is used to configure interfaces on the Spine
- **bridging role:** This role is used to configure all necessary VLANs and switch ports on the Spine
- **ipv4 role:** This role is used to configure the Spine's IP interfaces
- **bgp role:** This role is used to configure BGP protocol to allow the switches to be meshed together

All these reusable roles combined will be used to configure the Arista Spine switches and utilize the `eos_command` module heavily.

Similarly, a lot of the same modules can be utilized to configure the Leaf switches in the `leaf.yml` playbook, which targets the Leaf host group in the inventory and executes `common`, `interfaces`, `bridging`, `ipv4`, `bgp`, `ecmp`, and `m1ag` roles, as shown in the following screenshot:

```
---
- hosts: leaf
  gather_facts: no
  connection: local

  roles:
    - common
    - interfaces
    - bridging
    - ipv4
    - bgp
    - ecmp
    - mlag
```

The executed roles are used to carry out the following configuration:

- **common role:** This role is used to configure the IP routing table on the Spine
- **interfaces role:** This is used to configure interfaces on the Spine
- **Bridging role:** This is used to configure all necessary VLANs and switch ports on the Spine
- **ipv4 role:** This role is used to configure the Spine's IP interfaces
- **bgp:** This is used to configure BGP protocol to allow the switches to be meshed together
- **ecmp:** This is used to ensure equal cost multipathing is configured in the Leaf-Spine topology
- **mlag:** This is used to configure the switches redundantly at the top of the rack using mlag

This shows that roles can be reused if they are kept granular enough, with `var` files providing the necessary configuration changes to the roles, so it is important to avoid any hardcoded values.

The Leaf-Spine build out is a day one playbook, but why should a network engineer be interesting in taking all this time to set this up when it will only be used once? This, of course, is a common misconception as playbooks and roles have described the whole desired state of the network, and once the initial roles are written, going forward they can be used to mutate the desired state of the network at any point in the future.

The Ansible playbooks and roles could also be used to build the second data center in the same way, used as a disaster recovery solution, help to mutate the state if a data center re-IP is required, or even scale out more Spine and Leaf switches in the data center.

Taking the last example, in terms of scaling out a data center, this would be as simple as adding more Spine or Leaf switches to the Ansible inventory. Once the additional Arista switches have been zero touch provisioned after being racked and cabled by a data center operations team.

The network operator would then only need to make a small update to the `var` files to specify the VLANs that need to be used and update the inventory.

In the following example, the infrastructure is scaled to 15 Spine switches and 44 Leaf switches by modifying the inventory file:

```
[spine]
spineswitch[1-15]

[leaf]
leafswitch[1-44]
```

Although this is a pretty extreme scale out example, it should highlight the point and benefits of investing in automation. As such a scale out would take a network engineer weeks, whereas Ansible can carry out the same operations in minutes once the initial roles have been built out.

It really is worth the investment, this also means that the switches are built out consistently the same way as all the other switches, which alleviates manual error and makes the delivery of network changes more precise. Some people believe that automation is all about pace, but in networking, it should really be about consistency.

The same `spine.yml` and `leaf.yml` playbooks could also be executed against existing switches during the scale out, as Ansible is idempotent by nature, meaning only state changes will be pushed to the switches if the configuration has changed. If roles are not idempotent, then the modules being called are at fault.

This idempotency means the same day one playbook forming a `site.yml` that calls both `spine.yml` and `leaf.yml` could be run over existing switches and not change any configuration and be re-used without having to target just the changed switches. It is important to note that all Ansible changes should be tested against a test environment before being run in production.

Change requests

Network engineers despite this automation still need a separate process for manual change requests, right? The simple answer is no, manual changes would break the desired state that has been described in the day one playbooks. All network changes going forward should be pushed through the same configuration mechanism; there should be no such thing as a separate stream of work or an ad hoc command.

Making changes outside the process will only serve to break the Ansible playbooks and roles that were used to maintain the desired state and break the automation. It is important to note that utilizing network automation is an all or nothing approach that needs to be adopted by all team members and no changes should be done outside of the process or it breaks the model of repeatability and reliable changes. If features are lacking, the day one playbooks should be extended to incorporate the changes.

Self-service operations

With the use of Ansible for network operations, one of the typical bottle necks is the reluctance for network engineers to give development teams access to carry out network changes themselves, so this places a bottle neck on networking teams as usually a company will have more developers than network engineers.

This reluctance is because network changes are traditionally complex and a developer's forte is to develop code and create applications, not log onto networking devices to make firewall changes for their application.

However, if network engineers created a self-service playbook that defined a safe set of workflow actions, then developers could use it to interface with network devices in a safe way, this opens up a whole world of opportunity to remove that bottleneck.

This puts network engineers in the position of a **subject matter expert (SME)** role to help architect and use their network experience to create network automation that embodies networking best practices, to serve the needs of development teams.

This is instead of network engineers carrying out manual actions such as opening firewall ports manually when a developer raises a ticket. It is of course a change in role, but an automated approach is the way the industry is evolving.

Take the example of a firewall request, a development team have created a new application and need a test environment to deploy it in. When configuring the test environment, it needs networking, and a network engineer will ask the developer the ports they need to open in the firewall.

The developer doesn't know how to answer this question yet as they haven't finalized the application and want to start incrementally developing it in the test environment. Therefore, each time a new port needs to be opened, it means that a new network ticket is required to open the incremental port the development team discovers. This is not the optimum use of the network engineer or the developer's time as it causes frustration on both sides. A network engineer's time is better spent optimizing the network or adding improved alerting, not processing tickets to open firewall ports.

Instead Ansible could be used to create a self-service file. A developer could create a jinja2 template that could be checked into source control that lists the configuration file used to make firewall changes using the `template:` module. This shows the existing firewall line items and is available to developers to add new line items and submit a pull request to open a port on the firewall.

The network engineer then reviews the change and approves or rejects it. Ansible upon approval can be automatically triggered to push the change to a test environment; this makes sure that the config is valid.

In the following example, we see the playbook that replaces the `firewall.config` file with the updated jinja2 `firewall.j2` template and then reloads the firewall configuration from the new template:

```
tasks:
- name: Replace firewall module
  template: src=/firewall_template/firewall.j2 dest=/etc/firewall.conf owner=bin group=admin mode=0644
- name: Reload config
  fw_config: state=reload
```

This allows network teams to enable a self-service model. This speeds up the pace of network changes. It also removes the networking team as the bottleneck and pushes them to create appropriate tests and controls for network changes.

Self-service doesn't mean network engineers are no longer required. This means that they become the gatekeepers of the process instead of constantly rushing to keep up with the never ending chain of ad hoc requests they receive on a daily basis.

Summary

In this chapter, we looked at how Ansible can be used for server-side configuration management of network devices and looked at some of the industry leading network vendors, such as Arista, Cisco, and Juniper, who have all changed their operational models to use open standards and protocols that are well-suited to automation.

After reading this chapter, you should now be familiar with networking operating system from Cisco, Juniper, and Arista. The Ansible configuration management tool and concepts, such as Ansible Inventory, Ansible Modules, Ansible Playbooks, Ansible Roles, and Ansible var files and Jinja2 templates. Readers should also be familiar with Ansible Galaxy, the core Ansible modules available for network automation and methodologies to manage network devices using Ansible.

This chapter gave readers an understanding of use cases where tools such as Ansible can be used to automate everyday network operations that are carried out by network engineers. It should also give readers an insight into ways they could improve their network automation by utilizing configuration management tooling.

The key takeaways from this chapter are that configuration management tools such as Ansible now support network operations natively and vendors, such as Cisco, Juniper, and Arista, have created modules to facilitate automation of network operations. There is now no reason not to start automating network operations as these methods are fully supported by leading network vendors who understand that SDN operations are the future of network operations.

We have witnessed that Ansible is a very flexible tool. One of its main strengths is its ability to orchestrate APIs and help schedule software releases. Load balancing applications is a fundamental component of the software development release process, so in the the following chapter we will look at configuration management principles that can help orchestrate load balancers and help networking teams easily maintain complex load balancing solutions.

Useful links for Ansible network automation:

<https://www.youtube.com/watch?v=7FphWEFQbac>

<https://www.youtube.com/watch?v=VYEVjKvMKqU>

Useful links for Cisco:

<https://pynet.twb-tech.com/blog/automation/cisco-ios.html>

<http://www.cisco.com/c/en/us/support/switches/nexus-7000-series-switches/products-command-reference-list.html>



Useful links for Juniper:

https://www.juniper.net/documentation/en_US/junos15.1/topics/concept/junos-script-automation-overview.html

<http://www.juniper.net/techpubs/software/junos-security/junos-security10.4/junos-security-cli-reference/junos-security-cli-reference.pdf>

Useful links for Arista:

<https://www.arista.com/en/products/eos/automation>

<https://www.arista.com/docs/Manuals/ConfigGuide.pdf>

5

Orchestrating Load Balancers Using Ansible

This chapter will focus on some of the popular load balancing solutions that are available today and the approaches that they take to load balancing applications.

With the emergence of cloud solutions, such as AWS, Microsoft Azure, Google Cloud, and OpenStack, we will look at the impact this has had on load balancing with distributed load and centralized load balancing strategies. This chapter will show practical configuration management processes that can be used to orchestrate load balancers using Ansible to help automate the load balancing needs for applications.

In this chapter, the following topics will be covered:

- Centralized and distributed load balancers
- Popular load balancing solutions
- Load balancing immutable and static servers
- Using Ansible to orchestrate load balancers

Centralized and distributed load balancers

With the introduction of microservice architectures allowing development teams to make changes to production applications more frequently, developers no longer just need to release software on a quarterly basis.

With the move towards Continuous Delivery and DevOps, applications are now released weekly, daily, or even hourly with only one or a subset of those microservices being updated and released.

Organizations have found microservice architectures to be easier to manage and have moved away from building monolith applications. Microservice applications break a larger application into smaller manageable chunks. This allows application features to be released to customers on a more frequent basis, as the business does not have to redeploy the whole product each time they release. This means only a small microservice needs to be redeployed to deploy a feature. As the release process is more frequent and continuous, then it is better understood, normally completely automated, and ultimately load balanced.

Microservice architectures can also be beneficial for large businesses, which are distributed across many offices or countries as different teams can own different microservices and release them independently of one another.

This, of course, means that development teams need a way of testing dependency management, and the onus is put on adequate testing to make sure that a microservice doesn't break other microservices when it is released.

As a result, developers need to create mocking and stubbing services, so microservice applications can be effectively tested against multiple software versions without having to deploy the full production estate.

Creating a microservice architecture is a huge mindset shift for a business but a necessary one to remain competitive. Releasing monolithic applications is often difficult and time-consuming for an organization, and businesses that have quarterly release cycles will eventually lose out to competitors that can release their features in a quicker, more granular way.

The use of microservice architectures has meant that being able to utilize the same load balancing in test environments as production has become even more important due to how dynamic environments need to be.

So having test environments load balancing configuration as close to production environments as possible is a must. Configuration management tooling can be used to control the desired state of the load balancer.

The delegation of responsibilities also needs to be reviewed to support microservice architectures, so control of some of the load balancing provisioning should move to development teams as opposed to being a request to the network team to make it manageable and not to impede development teams. This, of course, is a change in culture that needs sponsorship from senior management to make the required changes to the operational model.

Load balancing requirements when using microservice applications will evolve as an application is developed or scaled up and down in size, so it is important that these aspects are made available to developers to self-service requests rather than wait on a centralized network team to make load balancing changes.

As a result of the shift towards microservices architectures, the networking and load balancing landscape has needed to evolve too to support those needs with PaaS solutions being created by many vendors to handle application deployment across hybrid cloud and load balancing.

Off-the-shelf PaaS solutions are a great option for companies that maybe aren't tech-savvy and are unable to create their own deployment pipelines using configuration management tooling, such as Chef, Puppet, Ansible, and Salt, to deploy their applications into cloud environments.

Regardless of the approach to deployment, roll your own or off-the-shelf PaaS. Both microservice and monolith applications still need to be supported when considering public, private, and hybrid clouds.

As a result, networking and load balancing need to be adaptable to support varied workloads. Although the end goal for an organization is ultimately a microservice architecture, the reality for most companies is having to adopt a hybrid approach catering to centralized and distributed load balancing methods to support both monolithic and cloud native microservices.

Centralized load balancing

Traditionally, load balancers were installed as external physical appliances with very complex designs and used very expensive equipment. Load balancers would be configured to serve web content with SSL requests terminated on the expensive physical appliances.

The load balancer would have complex configuration to route requests to applications using context switching, and requests would be served directly to the static backend servers.

This was optimal for monolith configurations as applications typically were self-contained and followed a three-tier model:

- A frontend webserver
- A business logic layer
- A database layer

This didn't require a lot of east to west traffic within the network as the traffic was north to south, traversing the frontend, business logic, and database. Networks were designed to minimize the amount of time taken to process the request and serve it back to the end user, and it was always served by the core network each time.

Distributed load balancing

With the evolution towards microservice architectures, the way that applications operate has changed somewhat. Applications are less self-contained and need to talk to dependent microservices applications that exist within the same tenant network, or even across multiple tenants.

This means that east-west traffic within the data center is much higher, and that traffic in the data center doesn't always go through the core network like it once did.

Clusters of microservices applications are instead instantiated and then load balanced within the tenant network using x86 software load balancing solutions with the endpoint of the microservices clusters **Virtual IP (VIP)** exposed to adjacent microservices that need to utilize it.

With the growing popularity of virtual machines, containers, and software-defined overlay networks, this means that software load balancing solutions are now used to load balance applications within the tenant network, as opposed to having to pin back to a centralized load balancing solution.

As a result load balancing vendors have had to adapt and produce virtualized or containerized versions of their physical appliances to stay competitive with open source software load balancing solutions, which are routinely used with microservices.

Popular load balancing solutions

As applications have moved from monoliths to microservices, load balancing requirements have undoubtedly changed. Today, we have seen a move towards open source load balancing solutions, which are tightly integrated with virtual machines and containers to serve east to west traffic between VPC in AWS or a tenant network in OpenStack as opposed to pinning out to centralized physical appliances.

Open source load balancing solutions are now available from **Nginx** and **HAProxy** to help developers load balance their applications or AWS elastic load balancing feature:

<https://aws.amazon.com/elasticloadbalancing/>

Just a few years ago, Citrix NetScalers (<https://www.citrix.com/products/netscaler-adc/>) and F5 Big-IP (<https://f5.com/products/big-ip>) solutions had the monopoly in the enterprise load balancing space, but the load balancing landscape has changed significantly with a multitude of new solutions available.

New load balancing start-ups such as Avi networks (<https://avinetworks.com/>) focus on x86 compute and software solutions to deliver load balancing solutions, which have been created to assist with both modern micro-service applications and monolith applications to support both distributed and centralized load balancing strategies.

The aim of this book is not about which load balancing vendor solution is the best; there is no *one size fits all* solution, and the load balancing solution chosen will depend on traffic patterns, performance, and portability that is required by an organization.

This book will not delve into performance metrics; its goal is to look at the different load balancing strategies that are available today from each vendor and the configuration management methods that could be utilized to fully automate and orchestrate load balancers which will in turn help network teams automate load balancing network operations.

Citrix NetScaler

Citrix NetScaler provides a portfolio of products to service an organization's load balancing requirements. Citrix provide various different products to end users, such as the **MPX**, **SDX**, **VPX**, and more recently the **CPX** appliances, with flexible license costs available for each product based on the throughput they support.

MPX and SDX are the NetScaler hardware appliances, whereas the VPX is a virtualized NetScaler and the CPX is a containerized NetScaler.

All of these products support differing amounts of throughput based on the license that is purchased (<https://www.citrix.com/products/netscaler-adc/platforms.html>).

All of the Citrix NetScaler family of products share the same common set of APIs and code, so the software is completely consistent. NetScaler has a REST API and a Python, Java, and C# Nitro SDK, which exposes all the NetScaler operations that are available in the GUI to the end user. All the NetScaler products allow programmatic control of NetScaler objects and entities that need to be set up to control load balancing or routing on MPX, SDX, VPX, or CPX.

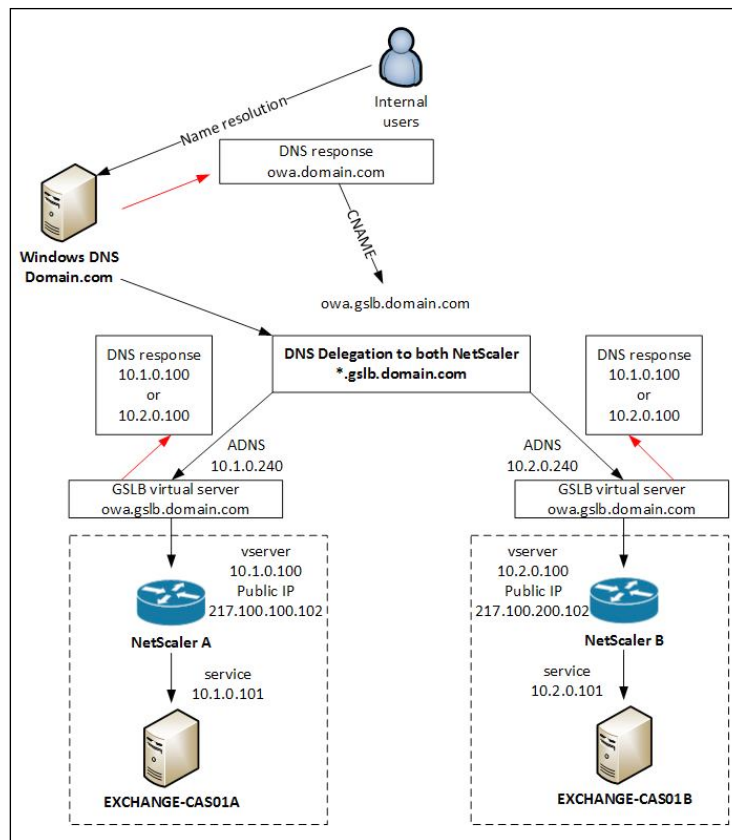
The NetScaler MPX appliance is a centralized physical load balancing appliance that is used to deal with a high number of **Transactions Per Second (TPS)**; MPX has numerous security features and complies with **Restriction of Hazardous Substances (RoHS)** and **Federal Information Processing Standard (FIPS)**, so the solution can be used by heavily regulated industries that require businesses to comply with certain regulatory standards.

MPX is typically used to do SSL offloading; it supports a massive amount of SSL throughput, which can be very useful for very highly performant applications, so the SSL offloading can be done on the hardware appliance.

MPX can be used to direct traffic to different tenant networks using layer 4 load balancing and layer 7 context switching or alternately direct traffic to a second load balancing tier.

The NetScaler SDX appliance is also a centralized physical appliance that is used to deal with a high number of TPS. SDX allows multiple VPX appliances to be set up as HA pairs and deployed on SDX to allow increased throughput and resiliency.

NetScaler also supports **Global Server Load Balancing (GSLB)**, which allows load to be distributed across multiple VPX HA pairs in a scale out model utilizing **CNAME**, which directs traffic across multiple HA pairs:



The VPX can be installed on any x86 hypervisor and be utilized as a VM appliance, and a new CPX is now available that puts the NetScaler inside a Docker container, so they can be deployed within a tenant network as opposed to being set up in a centralized model. All appliances allow SSL certificates to be assigned and used.

Every NetScaler appliance, be it MPX, SDX, VPX, or CPX, utilize IP the same object model and code that has the following prominent entities defined in software to carry out application load balancing:

- **Server:** A server entity on NetScaler binds a virtual machine or bare metal server's IP address to the server entity. This means the IP address is a candidate for load balancing once it is bound to other NetScaler entities.
- **Monitor:** The monitor entity on NetScaler are attached to services or service groups and provide health checks that are used to monitor the health of attached server entities. If the health checks, which could be as simple as a web-ping, are not positive, the service or service group will be marked as down, and NetScaler will not direct traffic to it.

- **Service group:** A service group is a NetScaler entity used to bind a group of one or more servers to an `lbserver` entity; a service group can have one or more monitors associated with it to health check the associated servers.
- **Service:** The service entity is used to bind one server entity and one or more monitor health checks to an `lbserver` entity, which specifies the protocol and port to check the server on.
- **lbserver:** An `lbserver` entity determines the load balancing policy such as round robin or least connection and is connected to a service group entity or multiple service entities and will expose a virtual IP address that can be served to end users to access web applications or a web service endpoints.
- **gslbserver:** When DNS load balancing between NetScaler appliances is required, a `gslbserver` entity is used to specify the `gslb` domain name and TTL.
- **csvserver:** The `csvserver` entity is used to provide layer 7 context switching from a `gslbserver` domain or `lbserver` IP address to other `lbserver`s. This is used to route traffic using the NetScaler appliance.
- **gslbservice:** The `gslbservice` entity binds the `gslbserver` domain to one or more `gslbserver`s entities to distribute traffic across NetScaler appliances.
- **gslbserver:** The `gslbserver` entities are is the `gslb`-enabled IP addresses of the NetScaler appliances.

Simple load balancing can be done utilizing the `server`, `monitor`, `service group`/`service`, and `lbserver` combination. With `gslbserver` and `csvserver`, context switching allows more complex requirements for complex routing and resiliency.

F5 Big-IP

The **F5 Big-IP** suite is based on F5's very own custom TMOS real-time operating system, which is self-contained and runs on Linux. TMOS has a collection of operating systems and firmware, which all run on BIG-IP hardware appliances or within the BIG-IP virtual instances. BIG-IP and TMOS (and even TMM) can be used interchangeably depending on the use case.

TMOS is at the heart of every F5 appliance and allows inspection of traffic. It makes forwarding decisions based on the type of traffic acting much in the same way as a firewall would, only allowing predefined protocols to flow through the F5 system.

TMOS also features iRules, which are programmatic scripts written using F5's very own **Tool Command Language (TCL)** that enables users to create unique functions triggered by specific events. This could be used to content switch traffic or re-order HTTP cookies; TCL is fully extensible and programmable and can carry out numerous operations.

The F5 Big-IP solution is primarily a hardware load balancing solution, that provides multiple sets of physical hardware boxes that customers can purchase based on their throughput requirements, and the hardware can be clustered together for redundancy.

The F5 Big-IP suite provides a multitude of products that provide services catering for load balancing, traffic management, and even firewalling.

The main load balancing services provided by the F5 Big-IP Suite are as follows:

- **Big-IP DNS:** F5's global load balancing solution
- **Local traffic manager:** The main load balancing product of the F5 Big-IP suite

The F5 Big-IP solution, like the Citrix NetScaler, implements an object model to allow load balancing to be programmatically defined and virtualized. F5 allows SSL certificates to be associated with entities.

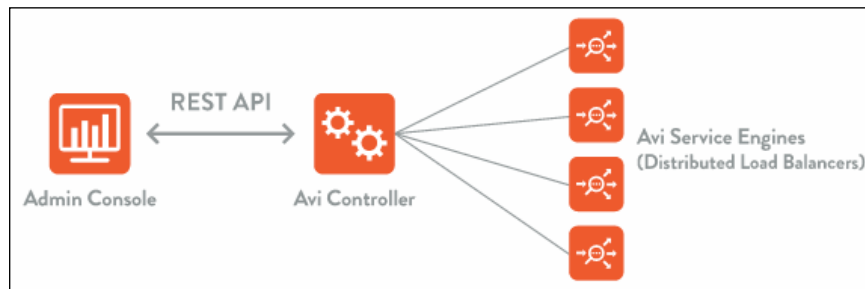
The following local traffic manager object entities allow F5 Big-IP to load balance applications:

- **Pool members:** The pool member entity is mapped to a virtual or physical server's IP address and can be bound to one or more pools. A pool member can have health monitors associated.
- **Monitor:** The monitor entity returns the status on specific pool members and acts as a health check.
- **Pool:** The pool entity is a logical grouping of a cluster of pool members that are associated; a pool can have health monitors associated with it as well as **Quality of Service (QoS)**.
- **Virtual servers:** The virtual server entity is associated with a pool or multiple pools, and the virtual server determines the load balancing policy, such as round robin or least connections. The F5 solution also will offer load balancing solutions based on capacity or fastest connection. Layer 7 profiles utilizing iRules can be configured against a virtual server and is used to expose an IP address to access pool members.
- **iRules:** iRules utilize the programmatic TCL, so users can author particular load balancing rules based on events such as context switching to different pools.

- **Rate classes:** Rate classes implement rate shaping, and they are used to control bandwidth consumption on particular load balancing operations to cap throughput.
- **Traffic classes:** Traffic class entities are used to regulate traffic flow based on particular events.

Avi Networks

Avi Networks are a relatively new start-up but have a very interesting load balancing product, which truly embraces the software-defined mandate. It is an enterprise software load balancing solution that comprises the **Avi Controller** that can be deployed on x86 compute. Avi is a pure software solution that deploys distributed Avi service engines into tenant networks and integrates with an AWS VPC and an OpenStack tenant:



The Avi Networks solution offers automated provisioning of load balancing services on x86 hypervisors, and it can automatically scale out to meet load balancing needs elastically based on utilization rules that users can configure.

The Avi Networks solution supports multiple or isolated tenants and has a real-time application monitoring and analytics engine that can work out where latency is occurring on the network and the location's packets are being routed from.

Avi also supports a rich graphical interface that shows load balancing entities so users have a visual view of load balancing, and it additionally supports anti-DDoS support.

All commands that are issued via GUI or API utilize the same REST API calls. The Avi Networks solution supports a Python and REST API. The Net Networks object model has numerous entities that are used to define load balancing in much the same way as NetScalers and F5:

- **Health monitor profile:** The health monitor pool profile entity specifies health checks for a pool of servers using health attributes.

- **Pool:** The pool entity specifies the IP addresses of virtual or physical servers in the form of a server list and has associated health monitor profiles; it also allows an event to be specified using a data script if a pool goes down. One or more pools are bound to the virtual service entity.
- **Custom policy:** The custom policy allows users to programmatically specify policies against a virtual service.
- **App profile:** The app profile entity allows each application to be modeled with associated http attributes, security, DDoS, caching, compression, and PKI attributes specified as part of the app profile associated with a virtual service.
- **Analytics profile:** The analytics profile makes use of the Avi analytics engine and captures threat, metrics, health score as well as latency thresholds and failure codes that are mapped to the virtual service entity.
- **TCP/UDP profile:** The TCP/UDP profile governs if TCP or UDP is used and any DDoS L3/L4 profiles are set.
- **SSL profile:** The SSL entity governs SSL ciphers that will be used by a virtual service entity.
- **PKI profile:** The PKI profile entity is bound to the virtual service entity and specifies the certificate authority for the virtual service.
- **Policy set:** The policy set entity allows users to set security teams to set policies against each virtual service governing request and response policies.
- **Virtual service:** The virtual service entity is the entry point IP address to the load balanced pool of servers and is associated with all profiles to define the application pools load balancing and is bound to the TCP/UDP, app, SSL, SSL cert, policy, and analytics profiles.

Nginx

Nginx (<https://www.nginx.com/>) supports both commercial and open source versions. It is an x86 software load balancing solution. Nginx can be used as both an HTTP and TCP load balancer supporting HTTP, TCP, and even UDP, and can also support SSL/TLS termination.

Nginx can be set up for redundancy in a highly available fashion using *keepalived*, so if there is an outage on one Nginx load balancer, it will seamlessly fail over to a backup with zero downtime.

Nginx Plus is the commercial offering and is more fully featured than the open source version, supporting features such as active health checks, session persistence, and caching.

Load balancing on Nginx is set up by declaring syntax in the `nginx.conf` file. It works on the principle of wanting to simplify load balancing configuration. Unlike NetScalers, F5s, and Avi Networks, it does not utilize an object model to define load balancing rules, instead Nginx describes load balanced virtual or physical machines as backend servers using declarative syntax.

In the following simple example, we see three servers, `10.20.1.2`, `10.20.1.3`, and `10.20.1.4`, all load balanced on port 80 using Nginx declarative syntax, and it is served on `http://www.devopsfornetworking.com/devops_for_networking`:

```
http {
    upstream backend {
        server 10.20.1.2;
        server 10.20.1.3;
        server 10.20.1.4;
    }

    server {
        listen 80;
        server_name www.devopsfornetworking.com;
        location / {
            proxy_pass http://devops_for_networking;
        }
    }
}
```

By default, Nginx will load balance servers using round-robin load balancing method, but it also supports other load balancing methods.

The Nginx `least_conn` load balancing method forwards to backend servers with the least connections at any particular time, whereas the Nginx `ip_hash` method of load balancing means that users can tie the same source address to the same target backend server for the entirety of a request.

This is useful as some applications require that all requests are tied to the same server using sticky sessions while transactions are processed.

However, the proprietary Nginx Plus version supports an additional load balancing method named `least_time`, which calculates the lowest latency of backend servers based on the number of active connections and subsequently forwards requests appropriately based on those calculations.

The Nginx load balancer uses a weighting system at all times when load balancing; all servers by default have a weight of 1. If a weight other than 1 is placed on a server, it will not receive requests unless the other servers on a backend are not available to process requests. This can be useful when throttling specific amounts of traffic to backend servers.

In the following example, we can see that the backend servers have load balancing method least connection configured. Server 10.20.1.3 has a weight of 5, meaning only when 10.20.1.2 and 10.20.1.4 are maxed out will requests be sent to the 10.20.1.3 backend server:

```
http {
    upstream backend {
        least_conn;
        server 10.20.1.2;
        server 10.20.1.3 weight=5;
        server 10.20.1.4;
    }

    server {
        listen 80;
        server_name www.devopsfornetworking.com;
        location / {
            proxy_pass http://devops_for_networking;
        }
    }
}
```

By default, using round-robin load balancing in Nginx won't stop forwarding requests to servers that are not responding, so it utilizes `max_fails` and `fail_timeouts` for this.

In the following example, we can see server 10.20.1.2 and 10.20.1.4 have the `max_fail` count of 2 and a `fail_timeout` of 1 second; if this is exceeded then Nginx will stop directing traffic to these servers:

```
http {
    upstream backend {
        server 10.20.1.2 max_fails=2 fail_timeout=1s;
        server 10.20.1.3 weight=5;
        server 10.20.1.4 max_fails=2 fail_timeout=1s;
    }

    server {
        listen 80;
        server_name www.devopsfornetworking.com;
        location / {
            proxy_pass http://devops_for_networking;
        }
    }
}
```

HAProxy

HAProxy (<http://www.haproxy.org/>) is an open source x86 software load balancer that is session aware and can provide layer 4 load balancing. The HAProxy load balancer can also carry out layer 7 context switching based on the content of the request as well as SSL/TLS termination.

HAProxy is primarily used for HTTP load balancing and can be set up in a redundant fashion using `keepalived` configuration using two apache configurations, so if the master fails, the slave will become the master to make sure there is no interruption in service for end users.

HAProxy uses declarative configuration files to support load balancing as opposed to an object model that proprietary load balancing solutions, such as NetScaler, F5 and Avi Networks, have adopted.

The HAProxy configuration file has the following declarative configuration sections to allow load balancing to be set up:

- **Backend:** A backend declaration can contain one or more servers in it; backend servers are added in the format of a DNS record or an IP address. Multiple backend declarations can be set up on a HAProxy server. The load balancing algorithm can also be selected, such as round robin or least connection.

In the following example, we see two backend servers, `10.11.0.1` and `10.11.0.2`, load balanced using the round-robin algorithm on port `80`:

```
backend web-backend
  balance roundrobin
  server netserver1 10.11.0.1:80 check
  server netserver1 10.11.0.2:80 check
```

- **Check:** Checks avoid users having to manually remove a server from the backend if for any reason, it becomes unavailable and this mitigates outages. HAProxy's default health always attempts to establish a TCP connection to the server using the default port and IP. HAProxy will automatically disable servers that are unable to serve requests to avoid outages. Servers will only be re-enabled when it passes its check. HAProxy will report whole backends as unavailable if all servers on a backend have failed their health checks.

A number of different health checks can be put against backend servers by utilizing the option `{health-check}` line item; for instance, `tcp-check` in the following example can check on the health of port `8080` even though port `443` is being balanced:

```
backend web-backend
  balance roundrobin
  option tcp-check
  server netserver1 10.10.0.1:443 check port 8080
  server netserver2 10.10.0.2:443 check port 8080
```

- **Access Control List (ACL):** ACL declarations are used to inspect headers and forward to specific backend servers based on the headers. An ACL in HAProxy will try to find conditions and trigger actions based on this.
- **Frontend:** The frontend declaration allows different kinds of traffic to be supported by the HAProxy load balancer.

In the following example, HAProxy will accept http traffic on port 80, with an ACL matching requests only if the request starts with `/network` and it is then forwarded to the `high-perf-backend` if the ACL `/web-network` is matched:

```
frontend http
  bind *:80
  mode http
  default_backend web-backend
  acl www.devopsfornetworking.com /web-network
  use_backend high-perf-backend if web-network
```

Load balancing immutable and static infrastructure

With the introduction of public and private cloud solutions such as AWS and OpenStack, there has been a shift towards utilizing immutable infrastructure instead of traditional static servers.

This has raised a point of contention with *pets versus cattle* or, as Gartner defines it *bi-modal* (<http://www.gartner.com/it-glossary/bimodal/>).

Gartner has said that two different strategies need to be adopted, one for new microservices, *cattle*, and one for legacy infrastructure, *pets*. *Cattle* are servers that are killed off once they have served their purpose or have an issue, typically lasting one release iteration. Alternately, *pets* are servers that will have months or years of uptime and will be patched and cared for by operations staff.

Gartner defines *pets* as Mode 1 and *cattle* as Mode 2. It is said that a *cattle* approach favors the stateless microservice cloud-native applications, whereas a *pet*, on the other hand, is any application that is a monolith, or potentially a single appliance or something that contains data, such as a database.

Immutable infrastructure and solutions such as OpenStack and AWS are said by many to favor only the *cattle*, with monoliths and databases remaining *pets* still need a platform that caters for long-lived servers.

Personally, I find the *pets* versus *cattle* debate to be a very lazy argument and somewhat tiresome. Instead of dumping applications into two buckets, applications should be treated as a software delivery problem, which becomes a question of stateless read applications and stateful applications with caching and data. Cloud-native microservice applications still need data and state, so I am puzzled by the distinction.

However, it is undisputed that the load balancer is key to immutable infrastructure, as at least one version of the application always needs to be exposed to a customer or other microservices to maintain that applications incur zero downtime and remain operational at all times.

Static and immutable servers

Historically, an operations team was used by companies to perform the following operations on servers:

- Rack and cable
- Providing firmware updates
- Configuring the RAID configuration
- Installing an operating system
- Patching the operating system

This was all before making the servers available to developers. Static infrastructure can still exist within a cloud environment; for example, databases are still typically deployed as static, physical servers, given the volume of data that needs to be persisted on their local disk.

Static servers mean a set of long-lived servers that typically will contain state.

Immutable servers, on the other hand, mean that every time a virtual machine is changed, a new virtual machine is deployed, complete with a new operating system and new software released on them, delete please. Immutable infrastructure means no in-place changes to a server's state.

This moves away from the pain of doing in-place upgrades and makes sure that snowflake server configurations are a thing of the past, where every server, despite the best intentions, has drifted slightly from its desired state over a period of time.

How many times when releasing software has a release worked on four out of five machines, and hours or days were wasted debugging why a particular software upgrade wasn't working on a particular server.

Immutable infrastructure builds servers from a known state promoting the same configuration to quality assurance, integration, performance testing, and production environments.

Parts of cloud infrastructure can be made completely immutable to reap these benefits. The operating system is one such candidate; rather than doing in-place patching, a single golden image can be created and patched using automation tooling such as Packer in a fully automated fashion.

Applications that require a caching layer are more stateful by nature so that cache needs to be available at all times to serve other applications. These caching applications should be deployed as clusters, which are load balanced, and rolling updates will be done to make sure one version of the cache data is always available. A new software release of that caching layer should synchronize the cache to the new release before the previous release is destroyed.

Data, on the other hand, is always persistent, so can be stored on persistent storage and then mounted by the operating system. When doing an immutable rolling update, the operating system layer can mount the data on either persistent or shared storage as part of the release process.

It is possible to separate the operating system and the data to make all virtual machines stateless, for instance, OpenStack Cinder (<https://wiki.openstack.org/wiki/Cinder>) can be utilized to store persistent data on volumes that can be attached to virtual machines.

With all these use cases considered, most applications can be designed to be deployed immutably through proper configuration management, even monoliths, as long as they are not a single point of failure. If any applications are single points of failure, they should be rearchitected as releasing software should never result in downtime. Although applications are stateful, each state can be updated in stages so that an overall immutable infrastructure model can be maintained.

Blue/green deployments

The **blue green deployment** process is not a new concept. Before cloud solutions came to prominence, production servers would typically have a set of servers consisting of blue (no live traffic) and green (serving customer traffic) that would be utilized. These are typically known as blue and green servers, which alternated per release.

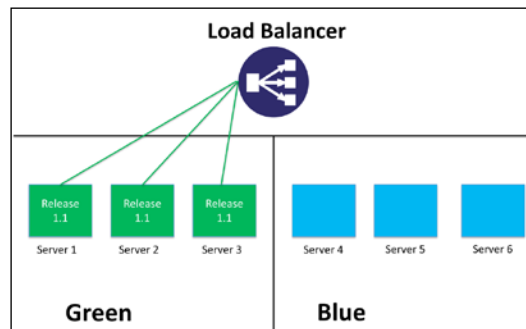
The blue green model in simple terms means that when a software upgrade needed to be carried out, the blue servers would be upgraded to the latest software version. Once the upgrade had been completed, the blue servers would become the new green servers with live traffic switched to serve from the newly upgraded servers.

The switching of live traffic was typically done by switching DNS entries to point at the newly upgraded servers. So once the DNS **Time To Live (TTL)** had propagated, end user requests would be served by the newly upgraded servers.

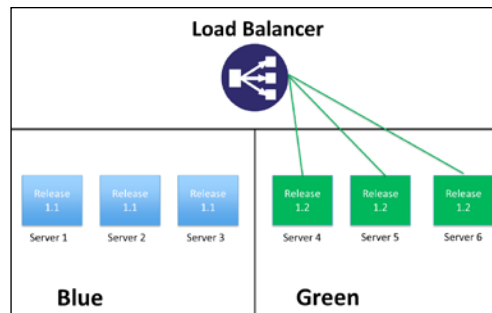
This means that if there was an issue with a software release, rollback could be achieved by switching back the DNS entries to point at the previous software version.

A typical blue green deployment process is described here:

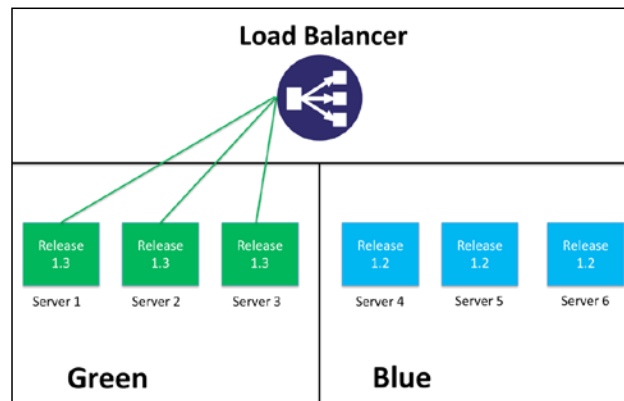
Release 1.1 would be deployed on servers 1, 2, and 3 and served on a load balancer to customers and made Green (live):



Release 1.2 would be deployed on servers 4, 5, and 6 and then be patched to the latest patch version, upgraded to the latest release and tested. When ready, the operations team would toggle the load balancer to serve boxes 4, 5, and 6 as the new production release, as shown later, and the previously green (live) deployment would become blue, and vice versa:

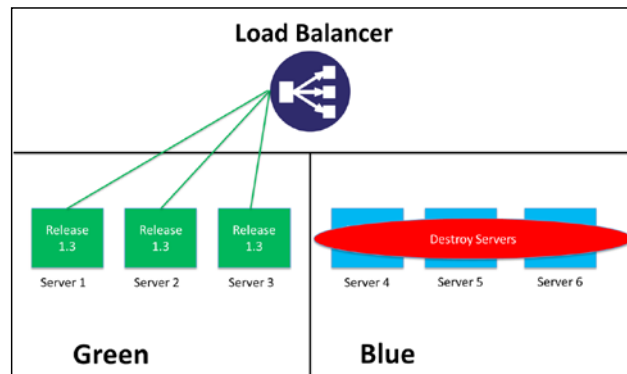


When the operations team came to do the next release, servers 1, 2, and 3 would be patched to the latest version, upgraded to **Release 1.3** from **Release 1.1**, tested, and when ready, the operations team would direct traffic to the new release using the load balancer, making **Release 1.2** blue and **Release 1.3** green, as shown in the following figure:



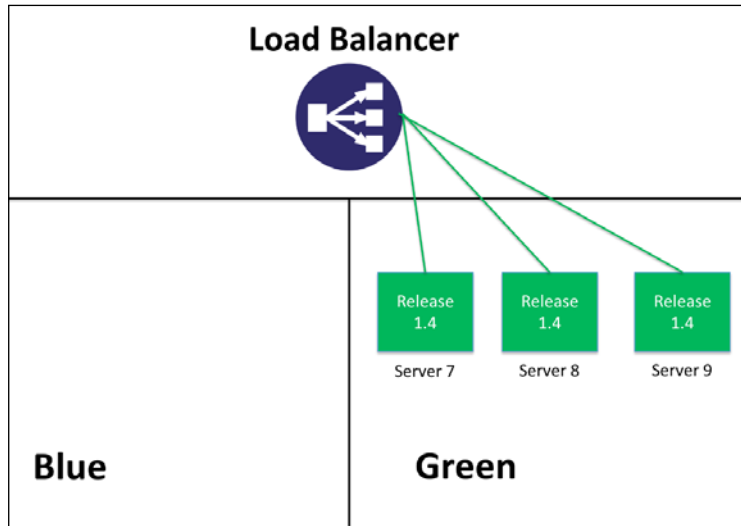
This was traditionally the procedure of running a blue green deployment using static servers.

However, when using an immutable model, instead of using long-lived static servers, such as Servers 1, 2, 3, 4, 5, and 6, after a release was successful, the servers would be destroyed, as shown here, as they have served their purpose:



The next time servers 4, 5, and 6 were required, instead of doing an in-place upgrade, three new virtual machines would be created from the golden base image in a cloud environment. These golden images would already be patched up to the latest version, so brand new servers 7, 8, and 9 with the old servers destroyed and the new **Release 1.4** would be deployed on them, as shown later.

Once server 7, 8, and 9 were live, servers 1, 2, and 3 would be destroyed as they have served their purpose:



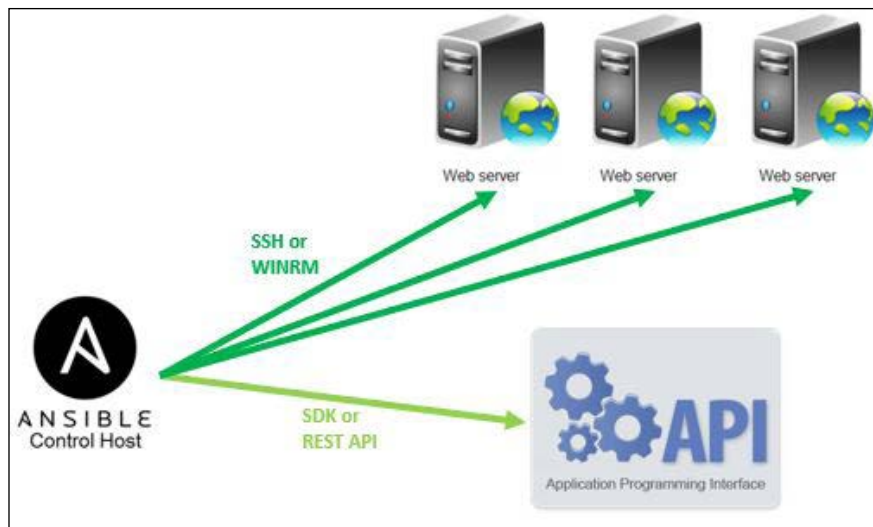
Using Ansible to Orchestrate load balancers

In *Chapter 4, Configuring Network Devices Using Ansible*, we covered the basics of Ansible and how to use an Ansible Control Host, playbooks, and roles for configuration management of network devices. Ansible, though, has multiple different core operations that can help with orchestrating load balancers, which we will look at in this chapter.

Delegation

Ansible delegation is a powerful mechanism that means from a playbook or role, Ansible can carry out actions on the target servers specified in the inventory file by connecting to them using SSH or WinRM, or alternately execute commands from the Ansible Control Host. WinRM is the Microsoft remote management standard and the equivalent of SSH for Windows that allows administrators to connect to Windows guests and execute programs.

The following diagram shows these two alternative connection methods with the Ansible Control Host either logging in to boxes using SSH or WinRM to configure them or running an API call from the Ansible Control Host directly:



Both of these options can be carried out from the same role or playbook using `delegate_to`, which makes playbooks and roles extremely flexible as they can combine API calls and server-side configuration management tasks.

An example of delegation can be found later where the Ansible extras HAProxy modules are used, with `delegate_to` used to trigger an orchestration action that disables all backend services in the inventory file:

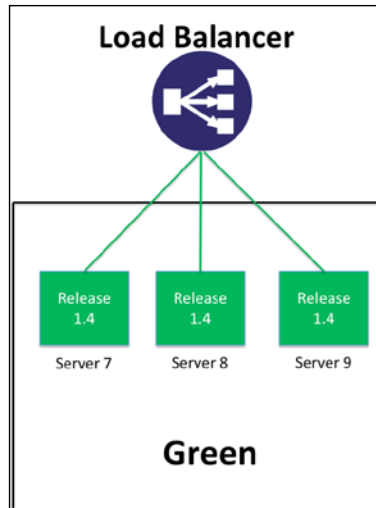
```
tasks:
- name: disable server in networking backend pool
  haproxy: state=disabled host={{ inventory_hostname }} backend=networking
  delegate_to: 127.0.0.1
```

Utilizing serial to control roll percentages

In order to release software without interruptions to service, a zero downtime approach is preferable, as it doesn't require a maintenance window to schedule a change or release. Ansible supports a *serial* option, which passes a percentage value to a playbook.

The *serial* option allows Ansible to iterate over the inventory and only carry out the action against a percentage of the boxes, completing the necessary playbook, before moving onto the next portion of the inventory. It is important to note that Ansible passes inventory as an unordered dictionary, so the percentage of the inventory that is processed will not be in a specific order.

Using the *serial* option that a blue/green strategy could be employed in Ansible, so boxes will need to be taken out of the load balancer and upgraded before being put back into service. Rather than doubling up on the number of boxes, three boxes are required, as shown in the following image, which all serve **Release 1.4**:



Utilizing the following Ansible playbook, using a combination of `delegate_to` and `serial`, each of the servers can be upgraded using a rolling update:

```
---
- hosts: application1
  serial: 30%

  tasks:

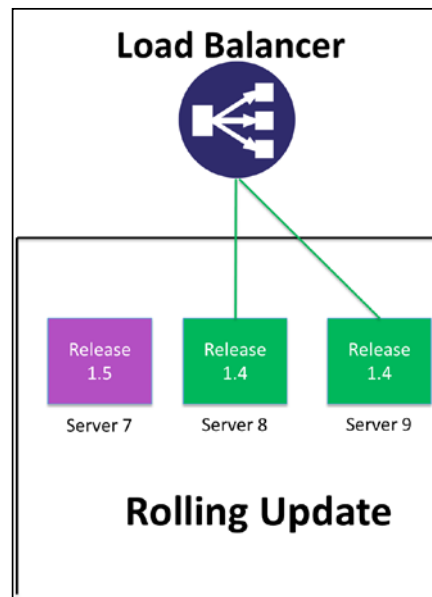
  - name: take out of load balancer pool
    haproxy: state=disabled host={{ inventory_hostname }} backend=backend_nodes
    delegate_to: 127.0.0.1

  - name: actual steps would go here
    yum: name=application1-1.5 state=present

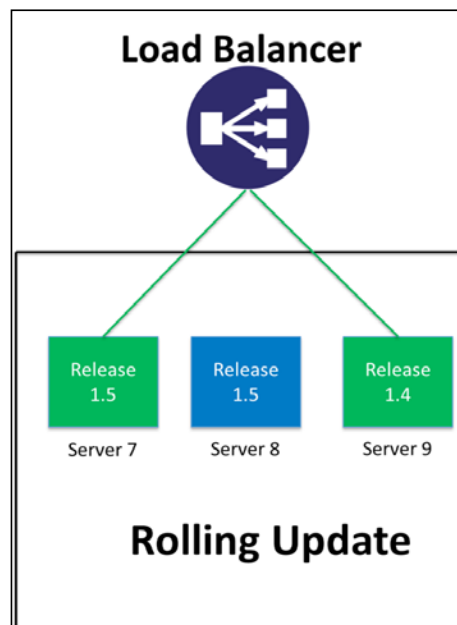
  - name: add back to load balancer pool
    haproxy: state=enabled host={{ inventory_hostname }} backend=backend_nodes
    delegate_to: 127.0.0.1
```

The playbook will execute the following steps:

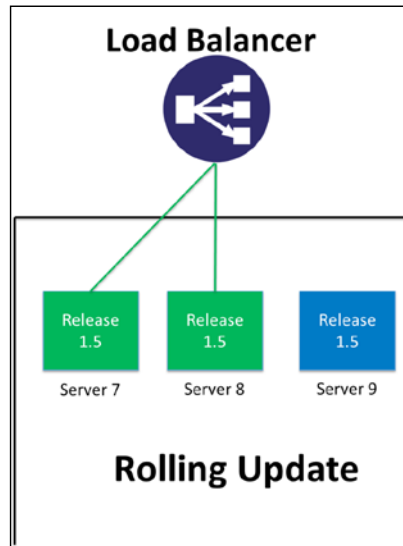
1. The `serial 30%` will mean that only one server at a time is upgraded. So, **Server 7** will be taken out of the HAProxy `backend_nodes` pool by disabling the service calling the HAProxy using a local `delegate_to` action on the Ansible Control Host. A `yum` update will then be executed to upgrade the server version new `application1` release **version 1.5** on **server 7**, as follows:



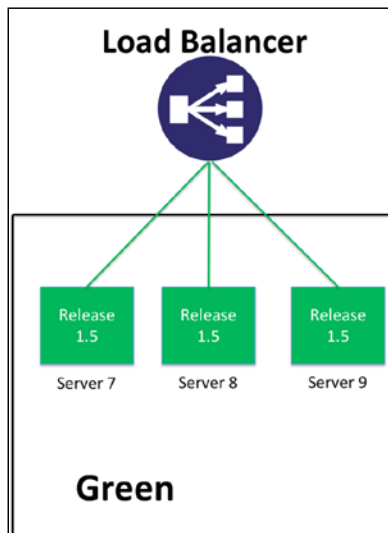
2. **Server 7** will then be enabled again and put into service on the load balancer using a local `delegate_to` action. The serial command will iterate onto `server 8` and disable it on HAProxy, before doing a `yum` update to upgrade the server version new `application1` release **version 1.5**, as follows:



- The rolling update will then enable **Server 8** on the load balancer, and the serial command will iterate onto **Server 9**, disabling it on HAProxy before doing a yum update, which will upgrade the server with the new `application1` release **version 1.5** alternating when necessary between execution on the local server and the server, as shown here:



- Finally, the playbook will finish by enabling **server 9** on the load balancer, and all servers will be upgraded to **Release 1.5** using Ansible as follows:



Dynamic inventories

When dealing with cloud platforms, using just static inventories is sometimes not enough. It is useful to understand the inventory of servers that are already deployed within the estate and target subsets of them based on characteristics or profiles.

Ansible has an enhanced feature named the dynamic inventory. It allows users to query a cloud platform of their choosing with a Python script; this will act as an autodiscovery tool that can be connected to AWS or OpenStack, returning the server inventory in JSON format.

This allows Ansible to load this JSON file into a playbook or role so that it can be iterated over. In the same way, a static inventory file can be via variables.

The dynamic inventory fits into the same command-line constructs instead of passing the following static inventory:

```
ansible-playbook -i inventories/inventory -l qa -e current_build=9
playbooks/add_hosts_to_netscaler.yml
```

Then, a dynamic inventory script, `openstack.py`, for the OpenStack cloud provider could be passed instead:

```
ansible-playbook -i inventories/openstack.py -l qa -e environment=qa
playbooks/devops-for_networking.yml
```

The dynamic inventory script can be set up to allow specific limits. In the preceding case, the only server inventory that has been returned is the quality assurance servers, which is controlled using the `-l qa` limit.

When using Ansible with immutable servers, the static inventory file can be utilized to spin up new virtual machines, whereas the static inventory can be used to query the estate and do supplementary actions when they have already been created.

Tagging metadata

When using dynamic inventory in Ansible, metadata becomes a very important component, as servers deployed in a cloud environment can be sorted and filtered using metadata that is tagged against virtual or physical machines.

When provisioning AWS, Microsoft Azure, or OpenStack instances in a public or private cloud, metadata can be tagged against servers to group them.

In the following example, we can see a playbook creating new OpenStack servers using the `os_server` OpenStack module. It will iterate over the static inventory, tagging each newly created group, and release metadata on the machine:

```
tasks:
  - os_server:
      state: present
      name: "{{ inventory_hostname }}"
      image: centos6
      flavor: 4
      nics:
        - net-name: network1
      meta:
        group: qa
        release: 9
```

The dynamic inventory can then be filtered using the `-l` argument to specify boxes with `group: qa`. This will return a consolidated list of servers.

Jinja2 filters

Jinja2 filters allow Ansible to filter a playbook or role, allowing it to control which conditions need to be satisfied before executing a particular command or module. There are a wide variety of different Jinja2 filters available out of the box with Ansible or custom filters can be written.

An example of a playbook using a Jinja2 filter would only add the server to the NetScaler if its metadata `openstack.metadata.build` value is equal to the current build version:

```
---
- hosts: application1
  serial: 30%

  tasks:
    - name: "add into load balancer pool"
      server_add_netscaler:
        state: present
        name: "{{ inventory_hostname }}"
        ns_proto: "http"
        delegate_to: 127.0.0.1
        when: openstack.metadata.build == {{ current_build }}
```

Executing the `ansible-playbook add_hosts_to_netscaler.yml` command with a `limit -1` on `qa` would only return boxes in the `qa` metadata group as the inventory. Then, the boxes can be further filtered at playbook or role using the `when` jinja2 filter to only execute the `add into load balancer pool` command if the `openstack.metadata.build` number of the box matches the `current_build` variable of 9:

```
ansible-playbook -I inventories/openstack.py -l qa -e environment=qa -e
current_build=9 playbooks/add_hosts_to_netscaler.yml
```

The result of this would be that only the new boxes would be added to the NetScaler `lbvserver` VIP.

The boxes could be removed in a similar way in the same playbook with a *not equal* condition:

```
- name: "remove from load balancer pool"
  server_add_netscaler:
    state: absent
    name: "{{ inventory_hostname }}"
    ns_proto: "http"
    delegate_to: 127.0.0.1
    when: openstack.metadata.build != {{ current_build }}
```

This could all be combined along with the serial percentage to roll percentages of the new release into service on the load balancer and decommission the old release utilizing dynamic inventory, delegation, jinja2 filters, and the serial rolling update features of Ansible together for simple orchestration of load balancers.

Creating Ansible networking modules

As Ansible can be used to schedule API commands against a load balancer, it can be easily utilized to build out a load balancer object model that popular networking solutions, such as Citrix NetScaler, F5 Big-IP, or Avi Networks, utilize.

With the move to microservice architectures, load balancing configuration needs to be broken out to remain manageable, so it is application-centric, as opposed to living in a centralized monolith configuration file.

This means that there are operational concerns when doing load balancing changes, so Ansible can be utilized by network operators to build out the complex load balancing rules, apply SSL certificates, and set up more complex layer 7 context switching or public IP addresses and provide this as a service to developers.

Utilizing the Python APIs provided by load balancing vendors, each operation could then be created as a module with a set of YAML `var` files describing the intended state of the load balancer.

In the example mentioned later, we look at how Ansible var files could be utilized by developers to create a service and health monitor for every new virtual server on a NetScaler. These services are then bound to the `lbserver` entity, which was created by the network team, with a roll percentage of 10%, which can be loaded into the playbook's `serial` command. The playbook or role is utilized to create services, lbmonitors, 34 servers and bind services to lbservers, whereas the var file describes the desired state of those NetScaler objects:

```
---
netscaler:
  lbserver:
    name: "devops_for_networking"
    subnet: "10.20.124.0/23"
    servicetype: "HTTP"
    lbmethod: "TOKEN"
    rule: HTTP.REQ.HEADER("x-ip").VALUE(0)
    persistencytype: "NONE"
    port: 80

    lbmonitor:
      monitorname: "mon-devops_for_networking"
      type: "HTTP-ECV"
      send: "GET /www/networking/v1.0/health"
      recv: "OK"
      lrtm: "ENABLED"
      downtime: 5

    service:
      servicetype: "HTTP"
      maxclient: 0
      port: 80

  roll_percentage: 10%
```

Summary

In this chapter, we saw that the varied load balancing solutions are available from proprietary vendors to open source solutions, and discussed the impact that microservices have had on load balancing, moving it from a centralized to distributed model to help serve east-west traffic.

We then looked at blue/green deployment models, the merits of immutable and static servers, and how software releases can be orchestrated using Ansible in either model. In the process, we illustrated how useful Ansible is at orchestrating load balancers by utilizing dynamic inventory, rolling updates, delegation, and jinja2 filters can all be used to help fulfill load balancing requirements.

The key takeaways from this chapter are that microservice applications have changed the way applications need to be load balanced, and distributed load balancing is better suited when deploying microservice applications, which have more east-west traffic patterns.

The reasons that immutable infrastructure is well-suited to microservice applications should now be clear.. The chapter also defined ways that state and data can be separated from the operating system and that different rolling update models are required to support stateless and stateful applications. In the next chapter, we will look at applying these same automation principles to SDN Controllers, primarily focusing on the Nuage solution. It will cover configuring firewall rules and other SDN commands, so the whole network can be programmatically controlled and automated.

6

Orchestrating SDN Controllers Using Ansible

This chapter will focus on SDN controllers and the ways they can enable network teams to simplify their daily tasks.

We will look at why SDN Controllers have been adopted and highlight some of the immediate business benefits they will bring if utilized correctly. It will focus on ways in which network operations need to be divided so network operations can scale, by utilizing automation.

This chapter will discuss the benefits of utilizing software-defined networking and look at practical configuration management processes that can be used to orchestrate SDN Controller APIs and object models. Finally, we will look at how Ansible can be used to execute and wrap some of these configuration management processes, using Nuage VSP as a practical example.

In this chapter, the following topics will be covered:

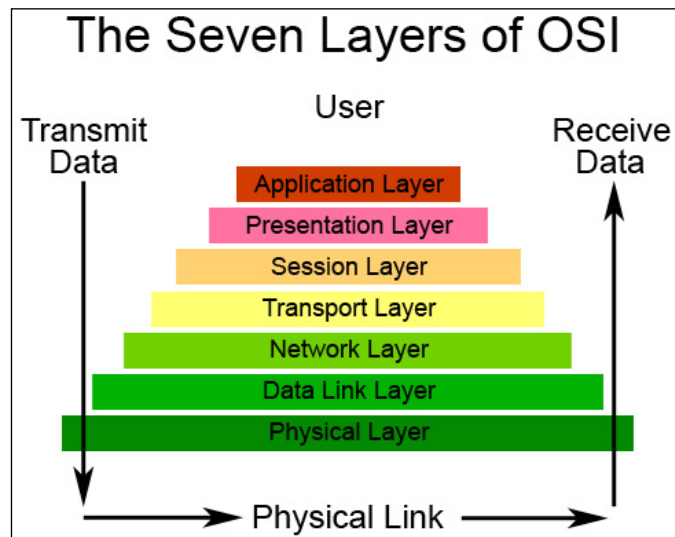
- Arguments against software defined networking
- Why would a company utilize SDN?
- Splitting up network operations
- Immutable networking
- Using Ansible to orchestrate SDN controllers

Arguments against software-defined networking

With the emergence of public clouds such as AWS, Microsoft Azure, and Google Cloud, networking is now being treated more like a commodity and has moved from silicon to software. This has allowed developers the ability to mutate the network to best serve the applications, rather than retrofit applications into an aging network, that is probably not optimized for modern microservice applications.

It would therefore seem nonsensical if any business would want to treat their internal data center networking any differently. However, like all new ideas, before acceptance and adoption comes fear and uncertainty, inherently co-related with the new or different ways of working.

Common arguments against using a clos Leaf-Spine architecture and SDN controllers center around one common theme, that it requires change and change is hard. We then harp back to the mythical 8th layer of the OSI model, and that is the **User** layer:



The network operators have to feel comfortable with any solution that is implemented. This is very important, but by the same token, the **User** layer is equally important as it is the networking service provided by the network team to end users. So ease of use is important on two levels, both network operations and the self-service operations provided to the consumers of the network.

Before a company considers putting in software-defined networking, they need to be doing it for the correct reasons and make it requirements-based. Simply implementing a new tool, in this case an SDN Controller, will not solve operational issues alone.

Organizations need to work out what the new operational model should be and utilize software-defined networking as a facilitator for those new business processes, focusing on speed of operations with the aim of removing networking as the bottleneck for application delivery. In short, network operations need to be DevOps friendly or they will inhibit software delivery and slow down the whole application lifecycle.

Added network complexity

Some of the arguments used against using overlay networks are that they are more complex than traditional layer 2 networks, with many more moving parts that could cause a bigger variety of failures.

Although the constructs of an overlay and underlay network may be different, it is fair to say software-defined networking is still a relatively new concept and a lot of people fear change. As long as the base requirements in terms of network availability, redundancy, performance, and speed of change are met, then there should be no reason not to implement software-defined networking.

The fear of software-defined overlay networks can be likened to operations staff's initial skepticism towards server virtualization when they initially argued against the introduction of hypervisors. These new concepts were initially viewed as an added layer of complexity and added abstraction layer that would probably not be as performant.

However, the portability and opportunities introduced by running a hypervisor greatly outweighed any performance implications for the vast majority of application use cases. The benefits included increased portability, flexibility, and speed of operations.

There are of course edge cases and some applications that don't fit into the virtualized model, but the benefits that virtualization brings for 99 percent of the data center mean that as a business solution it can't really be ignored.

Overlay networks give the same benefits to networking as hypervisors did to servers. Of course, when implementing a software-defined overlay network, the underlay should be built for redundancy, so that if a failure occurs, it occurs on the underlay and does not impact the overlay.

The underlay network should be horizontally scaleable and simple, in the case of a Leaf-Spine architecture, which has a series of Spine switches connected to Leaf switches that sit on top of each rack. The introduction of more racks paired with Leaf switches, or even a new Spine to prevent over-subscription of links, allow horizontal scalability.

On the topic of overlay networks adding complexity, Any systems reliability engineer or network engineer that has spent hours debugging an ill-performing link in a layer 2 Spanning Tree network will testify that Spanning Tree networks are themselves very complex by nature. The systems reliability engineer or network engineer will also probably be able to show you the network diagram they had to draw in an attempt to solve the issue as evidence of the complexity.

So networks are complex beasts at the best of times; however, when utilizing underlay and overlay networks, the main focus on the underlay network should be horizontal scalability and performance. It should ensure that network operators can easily scale out the network based on demand.

Alternatively, the focus of the overlay network is simplicity, so it should have easy-to-understand software constructs while at the same time ensure that the API endpoints can cope with the desired number of concurrent requests from consumers.

If implemented correctly, networks should be componentized into two distinct sections. The overlay user friendly software, much like AWS, Microsoft Azure, Google Cloud, or OpenStack, and the underlay is the gritty, hardcore, networking that needs to be well designed by a network architect and built for scale.

Lack of software-defined networking skills

Another argument against not implementing a software-defined network is lack of skills in the industry currently; with any new technology there is initially a lack of skilled people to support it. One viewpoint is that companies will have to hire completely new staff to implement software defined networking.

However, this can be offset by partnering with an SDN vendor or utilizing provided training programs for staff. It is a business transformation and as such, network staff will need to build new skills over a period of time.

But networking staff will need to evolve with the changes software-defined networking bring and build new skills like other teams in IT. Implementing software defined networking is a big change at first, but good networking staff should be excited and embrace these changes. The efficiency and benefits that can be had from implementing software-defined networking are undeniable.

Change can be daunting at first and can seem like a monumental cultural shift or effort at times. To initiate successful change in large or even small companies it usually has to come with top-down sponsorship or backing.

Adopting software-defined networking will mean changing the business's operational model and automation will need to be embraced at every level; network tasks in the overlay simply can't be manual when using an SDN controller. An organization implementing software-defined networking also needs to look at ways of automating the underlay. In this book we have already looked at ways in which APIs can be utilized to configure network devices, so really, both the underlay and overlay need to be automated.

The term software-defined data center is somewhat overused by vendors, but the principles behind it can't be ignored if a network team wants to provide a great user experience to the rest of the business. If a company puts in a software-defined networking solution as a standalone initiative, then it will add no true value if automation isn't written to speed up network operations utilizing the rich set of APIs that are provided. If companies are going to put in a software-defined network and have network engineers manually enter commands on network devices or use a GUI, the company may as well not bother, as they can do that with any out-of-the-box switch or router; they are wasting the opportunity a software-defined overlay network offers.

Just putting in the software-defined networking solution and still having developers raise network tickets will give zero business value; it will not increase efficiency, time to market, or the reliability of changes. To ensure organizations extract the significant business benefits out of software-defined networking, you need an all-or-nothing approach; network operations are either completely automated or over time become fragmented and broken.

If network engineers persist with doing manual updates outside the automated workflows, then it has the opportunity to break the whole operational mode. It changes the desired state of the network, and it could break the automation completely.

When putting in software-defined networking, automate all the common operations first and allow developers to serve themselves and make it immutable if possible. Being able to rebuild the network from source control management systems should be the aim as it acts as a record of change.

In *Chapter 3, Bringing DevOps to Network Operations*, we looked at ways of initiating cultural change. Humans are creatures of habit, they tend to stick with what they know; network engineers have spent years gathering networking certifications on ways to configure Spanning Tree algorithms and layer 2 networks, so this is a huge cultural shift.

Stateful firewalling to support regularity requirements

One of the main issues highlighted with software-defined networking has been the lack of stateful firewalling, due to Open vSwitch being based on flow data and being traditionally stateless. Until recently, reflexive rules were utilized to emulate stateful firewalling at the kernel user space level.

However, recent feature developments with Open vSwitch has allowed stateful firewalling to be implemented. So the lack of stateful firewalling is no longer an issue with Open vSwitch. **Connection tracking (conntrack)**, previously only available as part of iptables, has now been decoupled from iptables, meaning that it is now possible to match on connections as well as flow data.

The Nuage VSP platform has introduced stateful firewalling as part of its 4.x release. The Nuage VSP platform has replaced reflexive rules for stateful rules, to govern all ICMP and TCP ACL rules on the Nuage VRS (Nuage's customized version of Open vSwitch):

New Ingress Security Policy Entry

Name: Client to Webserver HTTP connections

Priority: Auto

Enable flow logging

Enable statistics collection

Traffic Type

Ether Type: IPv4 - 0x0800 Source Port: *

Protocol: TCP - 6 Destination Port: 80

DSCP Marker: Any Source IP Match: IP Address

Traffic Path

Origin Location: Subnet ClientNet (No description given)

Destination Network: Subnet WebNet (No description given)

Traffic Management

Action: Allow

Mirroring

No Mirroring

Stateful entry

Create

Why would organizations need software-defined networking?

Any good enterprise networks should be built with the following goals in mind:

- Performance
- Scalability
- Redundancy

The network, first and foremost, needs to be *performant* to meet customer needs. Customers can be end users in the data center or end users of the application in the public domain. With Continuous Delivery and deployment, if networking blocks a developer in a test environment, it is hampering a potential feature or bug fix reaching production, so it is not acceptable to have sub-standard pre-production networks and they should be designed as scaled-down functional replicas of production.

Scalability focuses on the ability to scale out the network to support company growth and demand. As more applications are added, how does the network horizontally scale? Is it cost effective? Can it easily be adapted to cater for new services such as third-party VPN access or point-to-point network integration? All these points need to be given proper consideration when creating a flexible and robust network design.

Redundancy is built on the concept that any enterprise network should have no single points of failure. This is so that the network can recover from a switch failure or an issue with a core router and not cause outages to customers. Every part of the network should be set up to maximize uptime.

These three points seem to have been the staple on which good networks were designed and built in the past. However, as applications have moved from monoliths to microservices, additional requirements are necessary for successful network operations.

Traditionally, monolithic applications have tended to have one setup operation and then remained fairly static, while microservice applications on the other hand have required more dynamic networks that are subject to greater variance of change.

The needs of the modern network have evolved and networks need to be updated rapidly to deal with the requirements of microservice architectures, without having to wait on a network engineer to process a ticket. With Continuous Delivery forming feedback loops, it is imperative that the process is quick and lean, and issues can be fixed quickly otherwise the whole process will break down and grind to a stand-still.

Software-defined networking adds agility and precision

Software-defined networking or in particular overlay networking, still focuses on *performance, scalability, and redundancy*; they should never be compromised, but also introduces the following benefits:

- Agility
- Mean time to recover
- Precision and repeatability

Software-defined networking puts the network into a software overlay network with associated object model, which allows the network to be programmable by exposing a rich set of APIs. This means that workflows can be used to set up network functions, the same way infrastructure can be controlled in a cloud or virtualization environment.

As the network is programmable, requesting a new subnet or making an ACL change can be done as quickly as spinning up a virtual machine on a hypervisor. Software-defined networking removes the traditional blockers or operational inhibitors. These have often included being required to raise a ticket to a network operation team to mutate the network, which was subject to a lengthy change control process. Instead, when utilizing software-defined networking, a developer can control a subnet of network operations via an API call so changes can be carried out at pace.

Mean time to recover has also improved when utilizing software-defined networking because network changes are programmable, so network inventory can be stored in source control management systems. This versions the network so any change is delivered via source control management and allows network changes to be modular, auditable, and easy to track.

If a breaking change has occurred to the overlay network, a version tree in the source control management system can be used to see what has changed since the network's last working release. The same programmable script can then be used to quickly roll back the network change back to the previous version and remove the issue. This is, of course, the beauty of implementing an immutable network rather than static networks, where the state is always as clean as the day one network and can be rolled forward or back on demand.

Repeatability in software-defined networking is catered for using programmatic operational workflows, so that all network changes are carried out in an identical way by all users. These operations can be executed using the API workflows approved by the network team against the overlay network.

The use of programmatic workflows means that network changes can be integrated into application deployment processes such as Continuous Delivery. This means network changes, like code, will be checked into source control management systems, pushed to a test environment using programmatic workflow actions (to manage the desired state of the network), tested and verified, and only then promoted onto the next test environment or production.

This repeatability of using an overlay network ensures all the constructs of a quality assurance test environment can be the same as a production environment, as all networking constructs are described in software and are easy to reproduce.

A good understanding of Continuous Delivery is key

Organizations looking to utilize software-defined networking should ideally already have a well-established Continuous Delivery model for code and infrastructure before tackling network operations. Companies committed to investing in a DevOps transformation would also benefit greatly from designing their new operational model around a software-defined network.

Companies which have mandated their business functions to automate all IT operations, inclusive of networking functions, would receive immeasurable quantifiable benefits from using an SDN controller to help their teams automate the network. Companies with an inherent understanding of DevOps, **continuous integration**, and **Continuous Delivery** are more likely to utilize SDN controllers to their full capabilities and drive innovation.

To emphasize the point, if overlay networks are modified by network engineers by hand rather than programmatically, it will bring no business value and the company will have missed the point.

Operational models need to change when implementing software-defined networking and if an issue occurs it needs to be built back into the automation to fix the issue so it doesn't re-occur. Any complex process, when initially automated, will probably hit some unexpected edge cases and fail under unexpected conditions. As a result, it is important that automated processes are continually iterated and improved on. Having teams adopt a continuous improvement methodology will ensure that automated processes are iterated and improved so they become more and more robust over time.

It is important to appreciate that edge cases will occur and to not panic when they do; fixing a problem with the automation fixes it for all users, but by the same token a problem with the automation can cause multiple users to be impacted, so it is a double-edged sword. Creating sufficient testing when creating automated processes to try and catch these edge cases in test environments becomes vitally important.

One of the benefits automation brings is that that all changes can be carried out with the precision of a highly skilled network engineer who can supply all their knowledge to automation. This means that every automated network change is done with the same care and precision as the best network engineer in the company.

The pre-approved and well-defined changes to automated workflows can be carried out by anyone in the company, not just the best engineer, if they are automated, so the bottleneck is removed from the network team freeing them up to work on more interesting tasks than the mundane repeatable **Business as Usual (BAU)** tasks that are more accurately done using automation.

Simplifying complex networks

Organizations that have very complex legacy networks would also be a prime candidate for benefiting from software-defined networking instead of fixing the existing network, which may not be possible due to having to adhere to 99 percent uptime targets. Instead, a new green-field network could be created in parallel with the existing network.

This will allow application workloads to be migrated to the new network over time and simplify the complexity of the existing network in the process. During the period of migration where both the new green-field network and old legacy network co-exist, the SDN overlay network can be used to route back to the legacy network for application dependencies that have yet to be migrated.

Another benefit of software-defined networking is that it allows private cloud solutions to run at increased scale. If private clouds are running more than 100 hypervisors, this is a scale at which an SDN solution would be of benefit, such as extending OpenStack Neutron capabilities to allow companies to run OpenStack at scale, as opposed to deploying multiple smaller OpenStack clouds to cope with bottlenecks.

Splitting up network operations

With the introduction of software-defined networking in a company or business there has to be a shift in operational responsibilities. If an organization runs multiple microservice applications, a fairly typical situation is that a company has 100 developers that develop those 200 microservices.

Each of those 200 microservices are combined together to deploy the company's customer facing website.

The company may use agile software development so each of the 100 developers are split into a set of delivery teams that contain 10 or so developers, each forming scrum teams, and each delivery team looks after a set amount of microservices relative to their complexity.

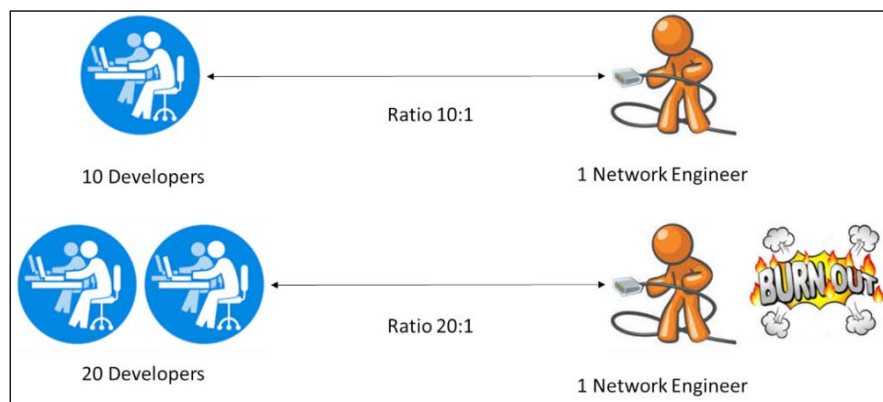
The company has 10 network engineers that are required to serve the networking needs of the 100 developers, as well as maintaining uptime of the network.

However, in this model, if all network operations are done manually, then the network engineers will not be able to keep up with the necessary change requests, so they will either have to work late nights and subsequently become burned out, so their productivity will drop. In this model, they are in reactive firefighting mode.

In this model, the productivity of the developers will probably be impacted too as the network engineers will become the bottleneck for throughput. The model described will simply not scale, so operational change is required.

In the scenario described, one network engineer will be required for every ten developers, and in future as the company expands it will want to invest in development staff to create more products. It is undoubtedly a harder sell for organizations to scale up their network teams to support those network operations, so network automation becomes a must in this scenario and the network team needs to work smarter.

Introducing new products and developers without changing the way a networking team operates can lead to burnout, so a network engineer will be able to support ten developers but not 20 when doing all network operations manually. Therefore, considering the developer to network engineer ratio is important when making the case for automation, as shown:



The business may then look at software-defined networking as the solution to solve their scaling problems, with the mindset of simplifying the network. This means that network engineers can carry out network changes more quickly to support developer demand.

But simply putting in a software defined networking solution such as CISCO ACI, Juniper Contrail, VMware NSX, or Nuage Networks will not help the situation unless processes are automated and the inefficient business processes are addressed.

New responsibilities in API-driven networking

The role of a network engineer in a software-defined network therefore has to evolve; they have to devolve some power to the developers like operations staff were required to for the creation of the infrastructure. But software-defined networking shouldn't mean giving complete, open access of the API to developer. This is also a recipe for disaster. Efficient controls need to be put in place that act as a quality gate, not as an inhibitor of productivity.

Some operational workflows in an overlay network should still be controlled by a qualified network engineer and governed by security, but not to the detriment of developer's productivity and requirements.

It wouldn't be fair to expect a developer to be well versed enough in networking to log onto a router and set up their routing requirements for their application unaided, so there has to be some middleground.

Allowing a developer access to network devices in an uncontrolled manner poses the risk of a network outage, which goes against one of the three main networking principles and compromises redundancy, and network engineers have a responsibility for uptime of the system.

Overlay architecture setup

When setting up an overlay network, it will normally be built in a green-field environment as part of an application migration program and target environment for a legacy network. The application migration could either be done in a piecemeal format or done in one step, where everything is migrated, then switched on as part of a migration big bang, go live activity.

Regardless of the application migration approach, it is very important that the overlay network is set up to achieve the following goals:

- Agility
- Minimize mean time to recover

- Repeatability
- Scalability

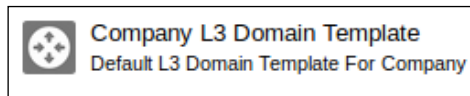
The performance of the network will be determined by the underlay components and silicon used, but the definition of the overlay network in terms of constructs and workflow of the SDN object model need to be correct to make sure that any operation can easily be carried out quickly, is repeatable, and that the design scales and can support roll-back. The SDN before implementation should be performance tested to make sure the virtualization overhead does not impact performance.

So, let's quickly recap on the Nuage VSP object model that was covered in *Chapter 2, The Emergence of Software-defined Networking*:

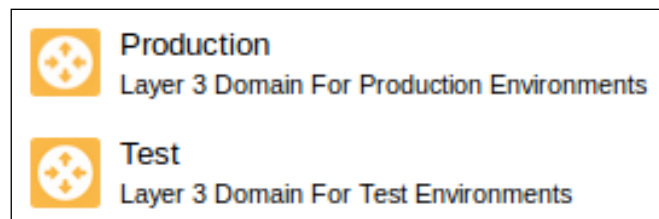
- **Organization:** Governs all layer 3 domains



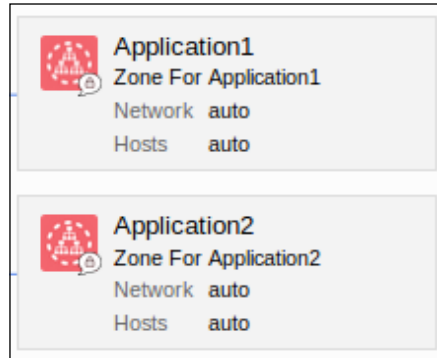
- **Layer 3 domain template:** A **Company L3 Domain Template** is required before child layer 3 domains are created. The **Company L3 Domain Template** is used to govern overarching default policies that will be propagated to all child layer 3 domains. If a **Company L3 Domain Template** is updated at template level, then the update will be implemented on all layer 3 domains that have been created underneath it, immediately.



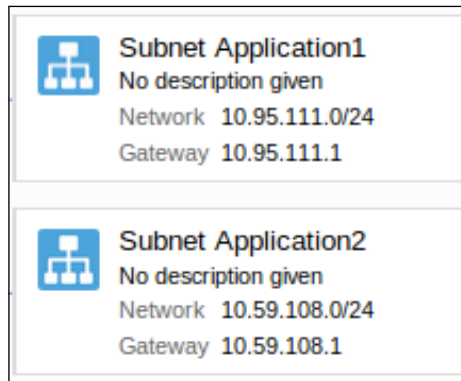
- **Layer 3 domain:** Can be used to segment different environments so users cannot hop from subnets deployed in a layer 3 **Test** domain to a layer 3 **Production** domain.



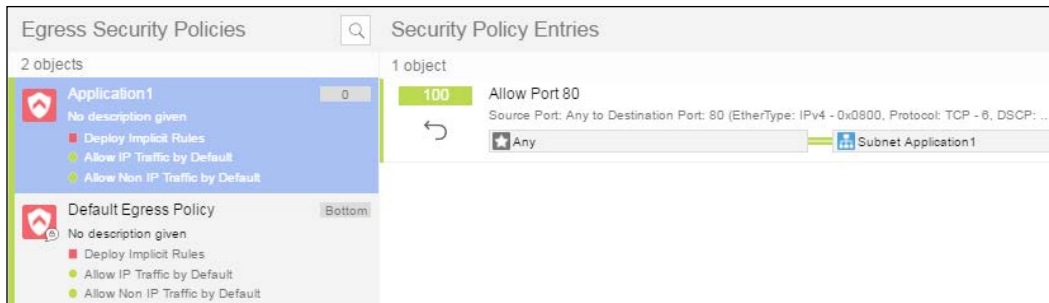
- **Zones:** A zone segment's firewall policies are at application level, so each micro-service application can have its own zone and associated Ingress and Egress policy per layer 3 domain.



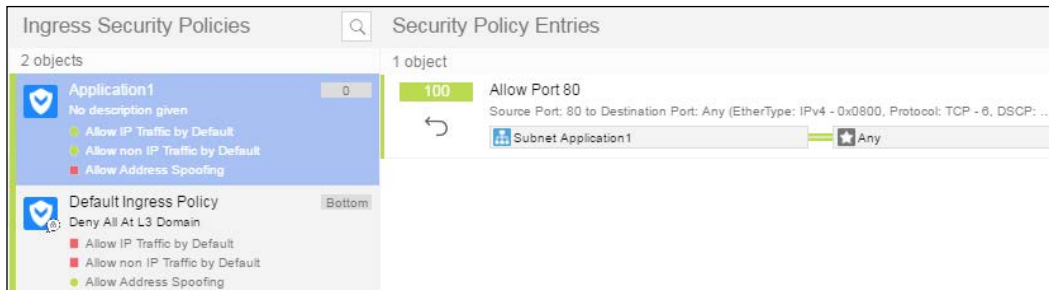
- **Layer 3 Subnet:** This is where VMs or bare-metal servers are deployed. In this example, we see **Subnet Application1** and **Subnet Application2**:



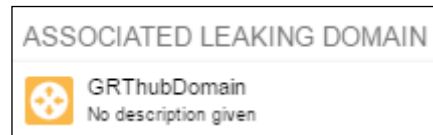
- **Application Specific Egress Policy:** Unique application policies for Egress rules that can be used to view each individual application's connectivity rules:



- **Application Specific Ingress Policy:** Unique application policies for ingress rules that can be used to view each individual application's connectivity rules:



- **Leaking Domain:** This is used to leak routes into the overlay network via a layer 3 subnet to bridge connectivity between the green-field network and a legacy network:



So, utilizing Nuage VSP as an example, we had an organization with two layer 3 domains dictating Test and Production, with a zone for each micro-service application encapsulating its unique micro-subnets and virtual machines:

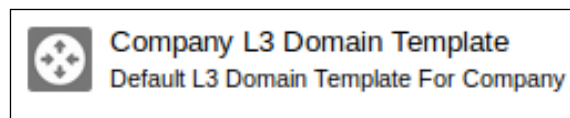


In terms of network setup, automation could be used by the network team and they would be in control of the following constructs in the overlay network:

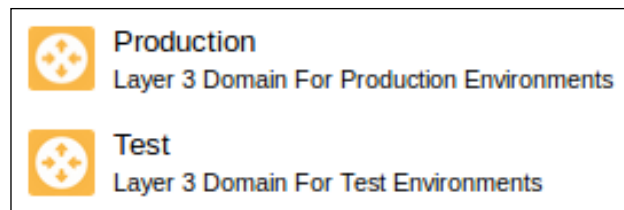
- **Organization:** Governs all layer 3 domains:



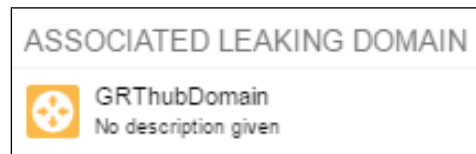
- **Layer 3 domain template:** Used to govern default policies:



- **Layer 3 domain:** Used to separate responsibilities between environments such as development and production:



- **Leaking Domain:** Used to make the legacy network accessible from the overlay network:



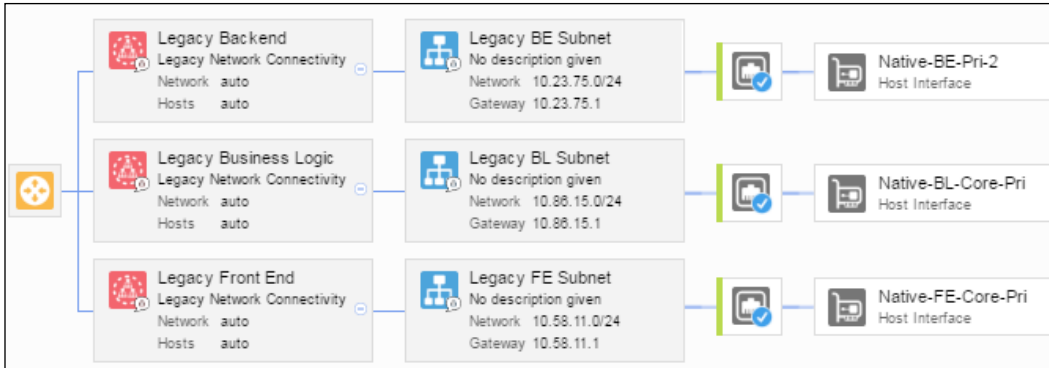
The organization is most likely a *day-one* setup activity, while the domain template policies can be defined and dictated by the network and security team. Any security policies applied across all networks, regardless of the domain they are deployed in, are governed by the domain template. So test environments will have identical template policies to production and meet all security, governance, and regularity requirements.

Development teams then have the ability to create unique test environments under the **Test** layer 3 domain with the same subsequent policies, without the need for the network team to audit each and every one. The application security rules that developers use can then be agreed between security and development teams without network teams having to become involved directly unless they are asked to advise on particular best practice ways of setting up ACL rules.

The other *day-one* setup activity will probably be setting up access to a legacy network that teams will be migrating applications from for a time, so they will still have dependent applications residing in that network.

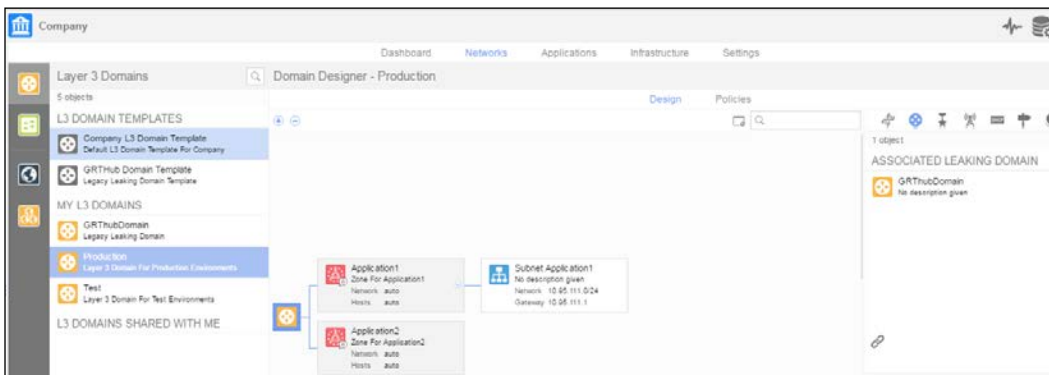
Nuage VSG, which is a hardware gateway device that connects external networks to the Nuage VSP platform and its associated leaking domain, can be used to do this. The Nuage VSG leaks routes from external networks into the overlay network and into specific layer 3 domains.

The Nuage VSP Platform allows network teams to define the **GRThubDomain** leaking domain in software that utilizes VSG. In this example, a leaking domain is set as IP host interfaces are connected into the **Front End, Business Logic** and **Back End** routers in the legacy network:



The Nuage VSP platform then allows the newly-created **GRThubDomain** to be associated with the **Production** or **Test** layer 3 domains by associating a leaking domain against them.

In the following example, the **GRThubDomain** leaking domain is associated with the **Production** layer 3 domain to allow legacy network routes to be accessible from zones and subnets residing under the **Production** layer 3 domain:



The network team will also be responsible for monitoring the network underlay and making sure that it is scaled out appropriately as more compute is introduced, so Leaf switches will be introduced and ordered as and when new racks are scaled out, while new Spine switches are introduced to avoid the saturation of links.

Self-service networking

It is important to focus on the network operations that developers typically require network tickets for as a start point. These are the common pain points for developers that prove to be blockers to productivity. Network operations can be effectively separated by looking at the common themes on network ticketing systems that have been raised by development teams.

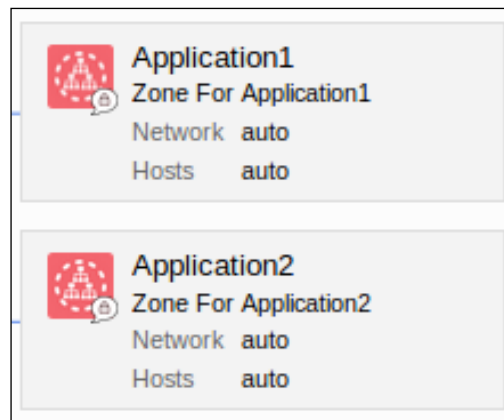
These are the more mundane BAU operations that network operators should make self-service:

- Opening firewall ports
- Creation of new development environments
- Connectivity to other applications

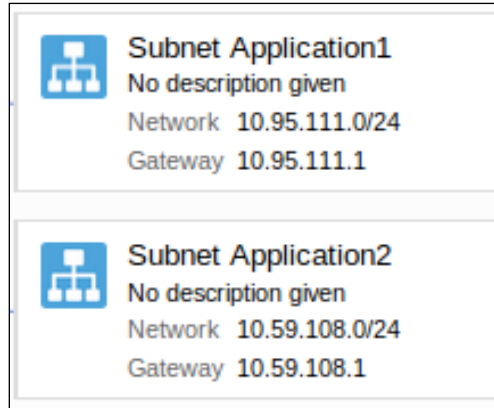
These operations should be set up as self-service operations in a software-defined network.

In terms of the Nuage VSP object model, network operators should allow developers the ability to control the following object model entities:

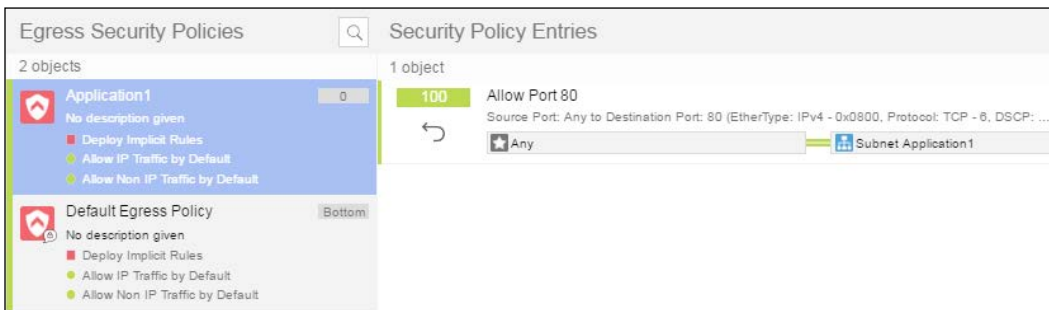
- **Zones:** They encapsulate a microservice application:



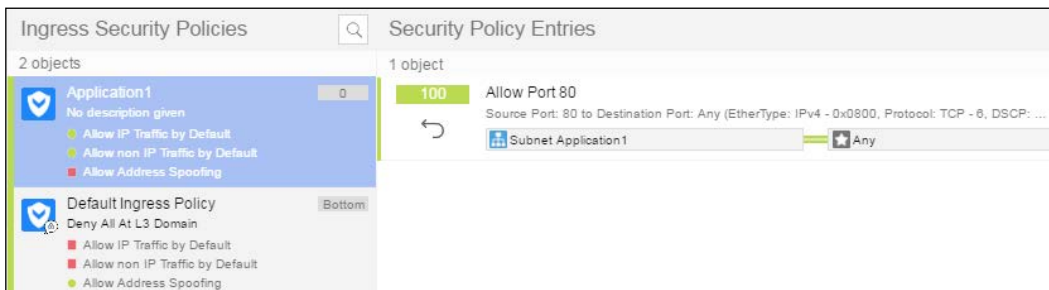
- **Layer 3 Subnet:** These define the IP range available to a microservice application



- **Application Specific Egress Policy:** This defines the Egress ACL policies for the microservice application:



- **Application Specific Ingress Policy:** This defines the Ingress ACL policies for the microservice application:



This will allow the network operations team to provide development teams with the organization, layer 3 domains, and the layer 3 domain template.

Underneath either the **Test** or **Production** layer 3 domains, development teams have the flexibility to create new zones unique to each microservice application, then any associated subnets and virtual machines that they need to provision.

The subnets will be micro subnets, so something akin to a /26, /27, or /28 may be acceptable. The network team will provide the subnet schema and a booking system where teams can reserve the address space in an IPAM solution if they are on-boarding an application or creating a new application, to prevent clashes with other teams.

As long as each delivery team follows those constructs, the networking team does not need to be involved in the provisioning of new applications or onboarding, it will become self-service, like AWS, Microsoft Azure, or Google Cloud.

However, in order to properly facilitate development teams, the network team should ideally create the self-service automation that the development teams can use to carry out the following in Nuage VSP along with the operations team:

- Creation of zones
- Deletion of zones
- Creation of subnets
- Deletion of subnets
- Creation of Ingress rules
- Deletion of Ingress rules
- Creation of Egress rules
- Deletion of Egress rules
- Creation of network macros (external subnets)
- Deletion of network macros (external subnets)

No matter the SDN solution implemented, the self-service constructs required will be similar, in order to scale network operations, a lot of the operations have to be automated and made self-service.

Ideally, these self-service workflow actions could be added to Ansible playbooks or roles and included in the deployment pipelines to provision the networking along with the infrastructure.

Immutable networking

To fully take advantage of the benefits of software-defined networking, utilizing immutable networking brings multiple benefits over static networking. Like infrastructure as code before it, networking as code and the utilization of immutable networking means that every time an application is deployed, its networking is freshly deployed from a source control management system that describes the desired state of the network. This means that network configurations don't drift over time.

Using a networking as code model to drive immutable networking allows application connectivity to be tested prior to production. Test environments mirroring production should be used to check application connectivity prior to releasing any network changes to production.

Implementing network changes as part of a Continuous Delivery model means that if application connectivity is proven to be wrong when it is tested in a test environment, then the application connectivity will be wrong in production environments. As a result, wrong connectivity changes should never reach production and should be caught prior to production by creating feedback loops that alert teams that the network change is not fit for purpose. Catching such issues will prevent outages and application downtime.

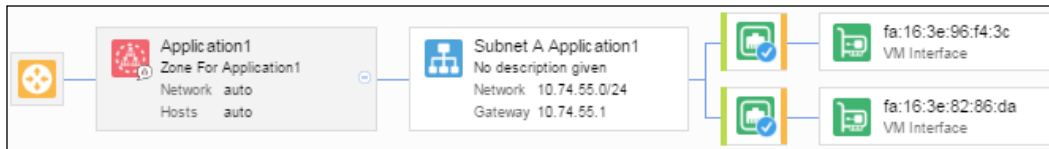
A/B immutable networking

Networking, as a result, should ideally be integrated and become part of the application release cycle, with networks being built from scratch every single release and loaded from the source control management system. Networks can be deployed using immutable A/B networking.

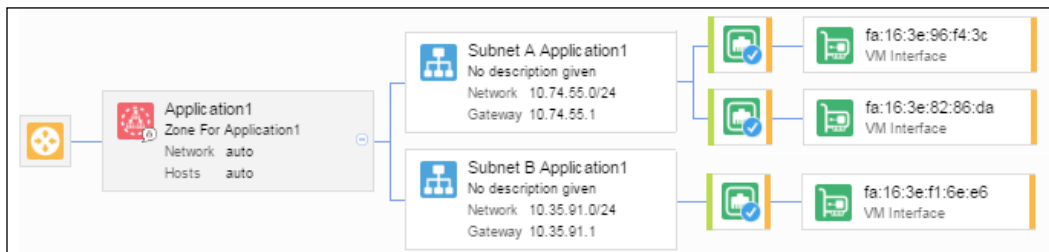
Using the Nuage VSP integrated with OpenStack as an example:

- A network will reside under a layer 3 domain
- Each zone will be unique to a particular microservice application
- Underneath the zone, a subnet will be created in both Nuage and OpenStack
- Virtual machines for each release will be created in OpenStack and associated with the Nuage subnet

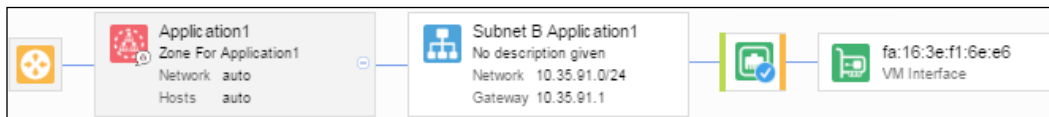
The first release of **Application1** version 1.1 is deployed to the **Test** layer 3 domain, deploying two virtual machines on **Subnet A Application1**, sitting under the **Application1** zone:



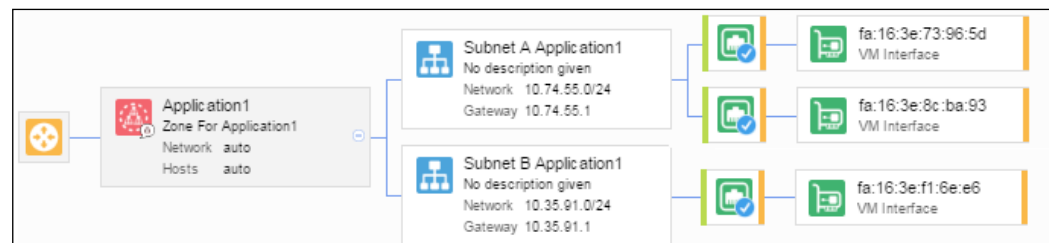
The second release of application version 1.2 is deployed to the **Test** layer 3 domain, scaling down the release and deploying one virtual machine on **Subnet B Application1**, sitting under the **Application1** zone:



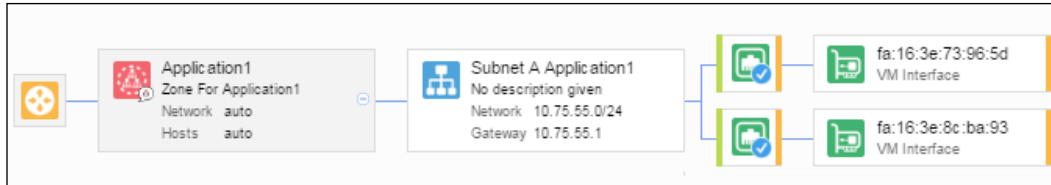
Once release 1.2 has been put into service on the load balancer, doing a rolling deployment, the new virtual machine on **Subnet B Application1** will be in service, **Subnet A Application1** can then be destroyed along with its virtual machines as part of the deployment clean-up phase:



The next release of **Application1**, release 1.3, will then be deployed into **Subnet A Application1**, and scaled up again to two virtual machines:



Once release 1.2 has been put into service on the load balancer, doing a rolling deployment, the new virtual machines on **Subnet A Application1** will be in service, **Subnet B Application1** can then be destroyed along with its associated virtual machine as part of the deployment clean-up phase:



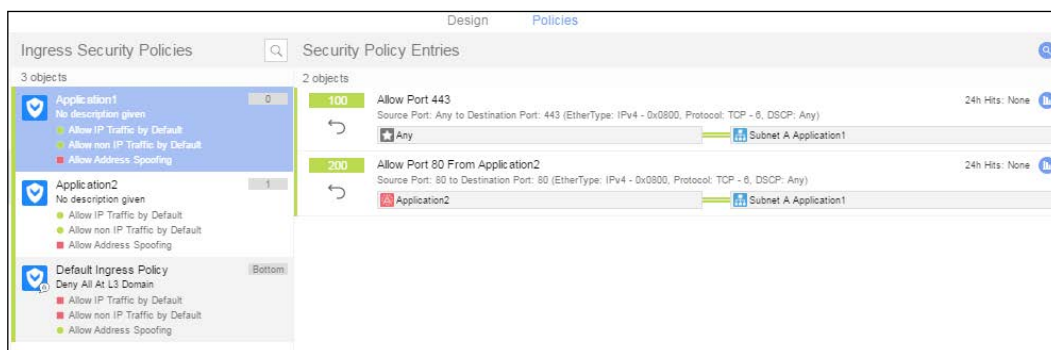
Releases will alternate between **Subnet A Application1** and **Subnet B Application1** for every release, building the network from source control each time and cleaning up the previous release each time.

The clean-up of redundant firewall rules

One of the major tech debt issues with firewalls is that over time they accumulate lots of out of date ACL rules as applications are retired or network connectivity changes. It is often a risk to do clean-up as network engineers are scared that they will potentially cause an outage. As a result, manual clean-up of firewall rules is required by the network team.

When utilizing A/B immutable network deployments, egress and ingress policies are associated with subnets, meaning that in Nuage VSP when a subnet is deleted, all ACL policies associated with that subnet will be automatically cleaned up too as part of the release process.

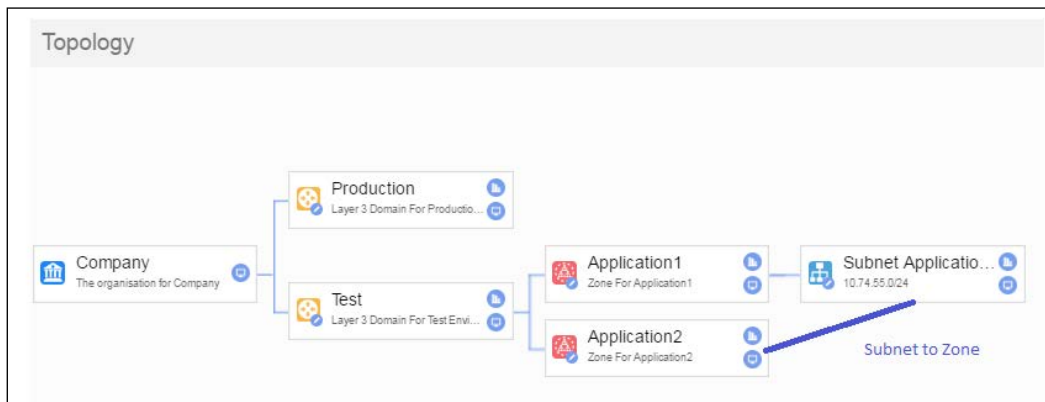
In the following example, **Subnet A Application1** has the following connectivity, so when the subnet is deleted as part of the release process, all these subnet-specific ACL rules will be cleaned up:



It is important to note that as ACL rules exist subnet to zone for application dependencies, if the A subnet deployment is in service, then the B subnet deployment will be brought up in parallel with its associated ACL Ingress and Egress rules to replace the A deployment.

All applications dependent on **Application1** will be required to have an ACL rule pointing at the zone rather than the subnet, this means they will not lose connectivity to the application as their rules will be zone-dependent rather than subnet-dependent. Having subnet to subnet rules would not work in an immutable subnet model.

To illustrate this, in the following example, currently deployed subnet **Application1** has a subnet to zone ACL rule to connect to **Application2**. So, despite **Application2** Egress and Ingress policies alternating between A and B deployments each time it is released as shown in the following diagram:



The required ACL rules are always available for **Application1** as a dependency as it subscribes to connectivity at the zone level as opposed to the subnet level:



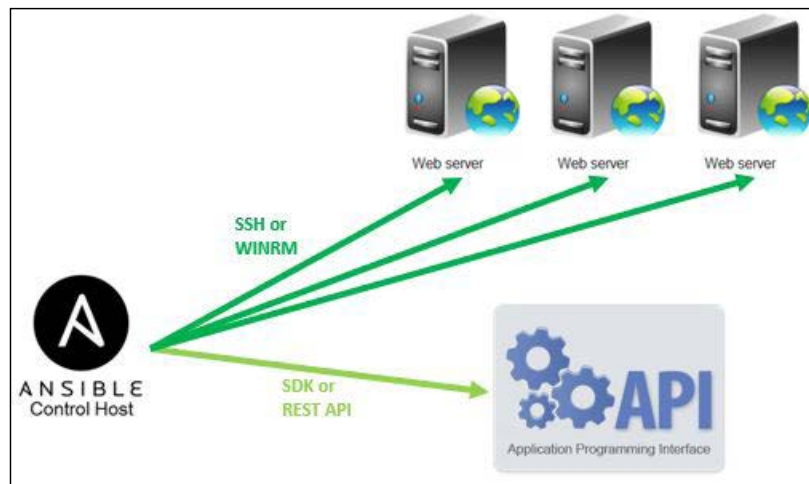
Application decommissioning

The use of immutable subnets makes the decommissioning of applications easy when they are no longer required. The clean-up logic already exists for subnets and associated ACL rules so that already-created automation can be re-used to do a full clean-up of the microservice application when it needs to be retired.

A clean-up pipeline can easily be provided by the operations and networking team for development teams to clean up applications that are no longer required. Their allocated subnet ranges can then be released by the IPAM solution so they are available to new microservice applications that need to be on-boarded onto the platform.

Using Ansible to orchestrate SDN controllers

Ansible, as discussed in *Chapter 5, Orchestrating Load Balancers Using Ansible*, can be used to issue and configure servers as well as issue commands directly to an SDK or REST API:



This is very useful when orchestrating SDN controllers that provide Restful API endpoints and an array of SDKs to control software-defined object models that allow network operators to automate all network operations.

In terms of the Nuage VSP platform, the VSD component, which builds the overlay network, is all REST API calls behind the scenes, so all operations can be orchestrated using the Nuage Java or Python SDK, which wrap REST API calls.

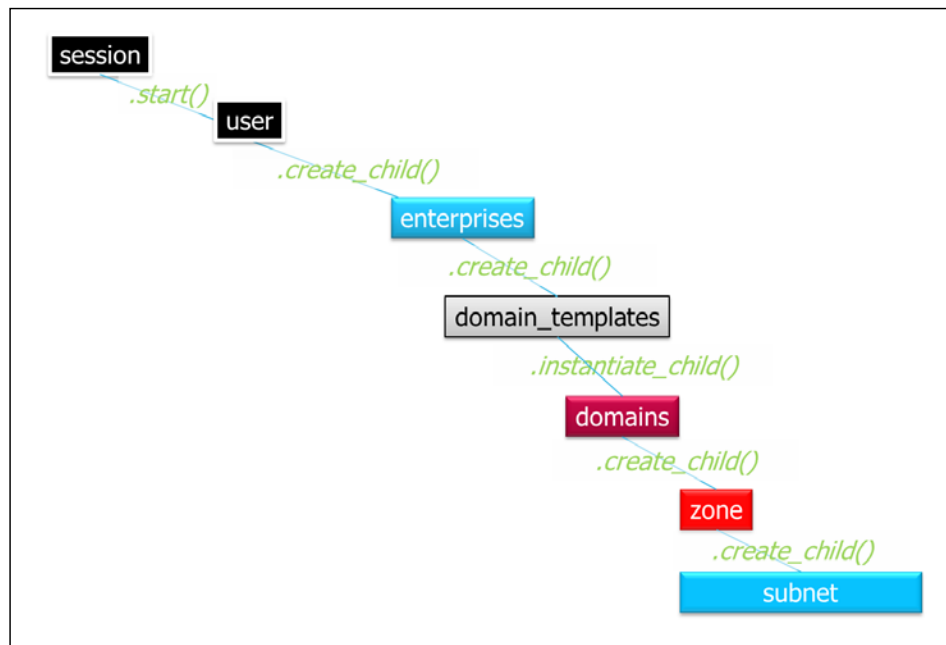
The Nuage VSPK SDK would simply need to be installed on the Ansible control host, and then it can be used to orchestrate Nuage. As Ansible is written in Python, modules can be easily created to orchestrate each object model in the Nuage entity tree.

Using the Nuage VSPK modules could alternately be written in any programming language that is available, such as Java, but Ansible's boilerplate for Python is probably the simplest way of creating modules.

The Nuage VSPK object model has parent and child relationships between entities, so lookups need to be done on parent objects to return the child entities using the unique identifier associated with the entity.

The following example highlights the list of operations required to build the Nuage VSPK object tree:

1. A new Nuage session is started.
2. A user is used to create a child enterprises.
3. A domain_templates is created as a child of the enterprise.
4. A domains is an instantiated as child of the domain template.
5. A child zone is created against the domain.
6. A child subnet is created against the zone.



Using SDN for disaster recovery

One of the main benefits of using Ansible for orchestration is that it can be used to create a set of day one playbooks to build out the initial network prior to it being used for self-service by developers. So the initial setup of the Nuage **organization**, **Company L3 Domain Template**, and layer 3 domains can be created among any other necessary operations as a day one playbook or role.

The Nuage Python VSPK can be utilized to easily create the organization called **Company**, layer 3 domain template called **L3 Domain Template**, and two layer 3 domains called **Test** and **Prod** as per the Nuage VSPK object model, as shown in the following screenshot:

```
#Open a session with VSD
session = vsdk.NUVSDSession(username=csproot,password=vsd_pass,enterprise=csp,api_url="https://nuage:8443",version="3.2")

#Start the session and get user credentials
session.start()
user=session.user

#Create an organisation
Organization = vsdk.NUEnterprise(name="Company",description="Company Description")
user.create_child(Organization)

#Create a Template
domain_template = vsdk.NUDomainTemplate(name="L3 Domain Template")

#Create Test domain
Organization.create_child(domain_template)
domain_test = vsdk.NUDomain(name="Test")
Organization.instantiate_child(domain_test,domain_template,commit=True)

#Create Production Domain
Organization.create_child(domain_template)
domain_prod = vsdk.NUDomain(name="Production")
Organization.instantiate_child(domain_prod,domain_template,commit=True)
```

Each of these Python commands can easily be wrapped in Ansible to create a set of modules to create a day one playbook utilizing `delegate_to localhost`, which will execute each module on the Ansible control host and then connect to the Nuage APIs.

Each module, by default, should be written so that it is idempotent and detects if the entity exists, before issuing a `Create` command. If the entity already exists, then it shouldn't issue a `Create` command if the overlay network is already in the desired state.

The day one playbook can be used to build the whole network from scratch in the event of a disaster if the whole network needs to be restored. The day one playbook should be stored in source control. While each deployment pipeline will build the application zones, subnets, and virtual machines under the initially defined structure.

A leaking domain governing legacy network connectivity and leaking domain association can also be added to the day one playbook if required.

Storing A/B subnets and ACL rules in YAML files

Ansible can also be utilized to store self-service subnet and ACL rule information in `var` files that will be called from a set of self-service playbooks as part of each development team's deployment pipelines. Each application environment can be stored in a set of `var` files defining each of the A/B subnets.

A playbook to create A or B subnets would be used to run `delegate_to localhost` to carry out the creation actions against the Nuage VSD API.

The playbook would be set up to do the following things:

1. Create the zone, if one has not already been created.
2. Create the subnet in Nuage mapped to OpenStack using a subnet YAML file.
3. Apply ACL policies for Ingress and Egress rules to the policies applying them directly to the subnet.

As with the day one playbook, unique modules can be written for each of the VSPK commands; in this example, the Python VSPK creates a zone called **Application1** and a subnet called **Subnet A Application1**:

```
#Create a Zone in the domain
zone = vsdk.NUZone(name="Application1")
domain.create_child(zone)

#Create a Subnet in the zone
subnetA = vsdk.NUSubnet(name="Subnet A Application1",address="10.74.55.0",netmask="255.255.255.0",gateway="10.74.55.1")
zone.create_child(subnetA)
```

So these commands can also be wrapped in Ansible modules, should be completely idempotent, and the desired state of the network is determined by the `var` files that are stored in source control.

The logic in the playbook would load the `var` files by pulling them from source control at deployment time. The playbook would then use the Jinja2 filter conditions to detect if either the A or B subnet or neither was present using the `when` conditions.

If neither subnet was present, subnet A would be created, or if subnet A was present, then subnet B would be created.

The playbook could read this information from the environment specific `var` file that is specified in the following screenshot. As it is idempotent, it will run over the zone, creating it if it doesn't already exist, and use the `jinja2` playbook when conditions to either create subnet A or B:

```
---
layer3_domain: Test
zone: Application1
subnets:

  - name: Subnet A Application1
    address: 10.74.55.0/24
    gateway: 10.74.55.1

  - name: Subnet B Application1
    address: 10.35.91.0/24
    gateway: 10.35.91.1
```

A unique set of A and B subnets would be checked into source control as a prerequisite for every required environment, with one or more environments per layer 3 domain.

ACL rules should ideally be consistent across all environments encapsulated in a layer 3 domain, so an explicit set of ACL rules would be created and assigned to the application's unique policy for Ingress and Egress rules that would span all environments.

Each environment could have its own unique policy for Egress and Ingress per layer 3 subnet. The Ansible playbook could then append a unique identifier for the environment to the policy name if multiple environments existed under the **Test** layer 3 domain to server integration, UAT, or other test environments.

The unique ACL rules for an application can be filled in by development teams as part of the on-boarding to the new platform based on the minimum connectivity required to make the application function, with a deny all applied to the layer 3 domain template.

The ACL rules should always be subnet to zone for inter-dependencies and each ACL rule will be created with the subnet as the source, so that when subnets are destroyed, the ACL rules will automatically be cleaned up.

An example of how the self-service ACL rules file would look is displayed as follows It would create two ingress rules and one Egress rule against the **Application1** policy:

```
---
acl_rules:
  ingress:
    - name: ""
      protocol: "TCP"
      src_type: "ANY"
      src_port: "*"
      dst_port: 443
    - name: ""
      protocol: "TCP"
      src_type: "ANY"
      src_port: "*"
      dst_port: 80

  egress:
    - name: "native-dbs-1521"
      protocol: "TCP"
      dst_type: "Zone"
      dst: "Application2"
      dst_port: 80
```

The self-service playbook could be provided to development teams so that they always have a standard way to create zones and subnets. The YAML structure of the `var` files will also provide templates of what the desired state of the network should be. This means that pointing the automated pipelines at another Nuage endpoint would mean the whole network could be built out programmatically from source control.

Summary

In this chapter, we have looked at different networking operations that SDN controllers can help automate, and sought to debunk some of the common misconceptions associated with software-defined networking.

We then looked at ways in which companies can benefit from using software-defined networking and looked at ways in which SDN solutions can help solve some of the challenges associated with network operations.

The chapter then focused on ways that network operations need to adapt and embrace automation so development teams can self-serve a subset of different networking tasks, and ways in which networking can be divided and responsibilities shared. We then focused on the benefits of immutable A/B networking and how it can help simplify the network and build consistent programmatically controlled networks while keeping firewall rules clean.

In this chapter, you should have learned why software-defined networking is important to organizations looking to scale network operations. We have also covered ways in which overlay network object models can be utilized by microservice applications and the benefits of immutable networking and A/B subnets.

Key takeaways from this chapter also include different ways that SDN controllers can help network operators to build out day one networks, which pieces of network operations can be made self-service, and the ways in which Ansible can be used to programmatically control network operations using Rest API calls or an SDK.

In the next chapter, we will look at continuous integration and how network operations can take some of the best practices from development teams and apply them to networking operations, so that networking is versioned properly and can be used to roll forward and roll back changes.

Once we have established a basis for continuous integration, we will move onto chapters that cover network testing and Continuous Delivery, which will outline a set of best practices that should allow network teams to integrate network automation into deployment pipelines.

7

Using Continuous Integration Builds for Network Configuration

This chapter will focus on continuous integration, what the process entails, and why it is applicable to network operations. We will look at why continuous integration processes are vitally important when automating network operations.

This chapter will discuss the benefits of configuration management tooling and we will look at practical configuration management processes that can be used to set up continuous integration processes and tooling that is available to support continuous integration processes.

In this chapter, the following topics will be covered:

- Continuous integration overview
- Continuous integration tooling available
- Network continuous integration

Continuous integration overview

Continuous integration is a process used to improve the quality of development changes. A continuous integration process, when applied to developers, takes new code changes and integrates it with the rest of the code base. This is done early in the development lifecycle, creating an instant feedback loop and associated pass or failure against the change.

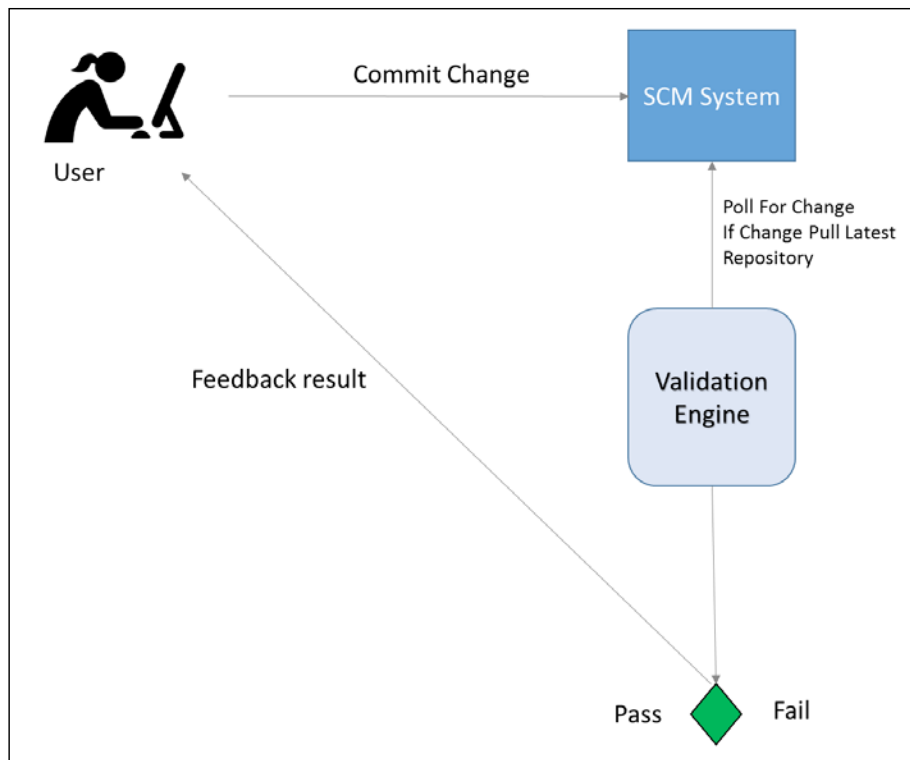
Within the remits of DevOps, continuous integration is a key component as it uses centralized tooling to make changes visible to other users and promotes collaboration and integration of changes earlier in the software development lifecycle. Continuous integration is often coupled with Continuous Delivery processes, where continuous integration is used as the first part of the software delivery lifecycle.

Prior to continuous integration being implemented, developers would sometimes only find out that code changes did not work when a release needed to be packaged. At this point, all developer changes were combined by a release management or operations team. By the time the release was ready to be packaged, a developer would have moved on to new tasks and not have been currently working on that piece of work anymore, meaning fixing the issue incurred more time delaying the release schedule.

A good continuous integration process should be triggered every time a developer commits a change, meaning that they have a prompt feedback cycle to tell them if their change is good, rather than finding out weeks or months later that their commit had an issue that will slow down the release process.

Continuous integration works on the premise of fixing as far left as possible, meaning at development time, with the furthest right being production. What this phrase really means is that if an issue is found earlier in the development cycle then it will cost less to fix and have less of an impact to the business as it will ideally never reach production.

A continuous integration process follows the following steps, **Commit Change to Source Control Management (SCM)**, the repository change is validated, and a pass or failure is issued back to the user:



The output of the continuous integration process should be what is shipped to test environments and production servers. It is important to make sure that the same binary artifacts that have been through continuous integration and relative testing are the same ones that will eventually be deployed onto the production servers.

Processes such as continuous integration are used to create feedback loops that show issues as soon as they occur, which saves cost. This means the change is fresh in the implementers mind and they will be able to fix it or revert the change quickly, with developers currently iterating the code collaboratively and fixing issues as soon as they occur.

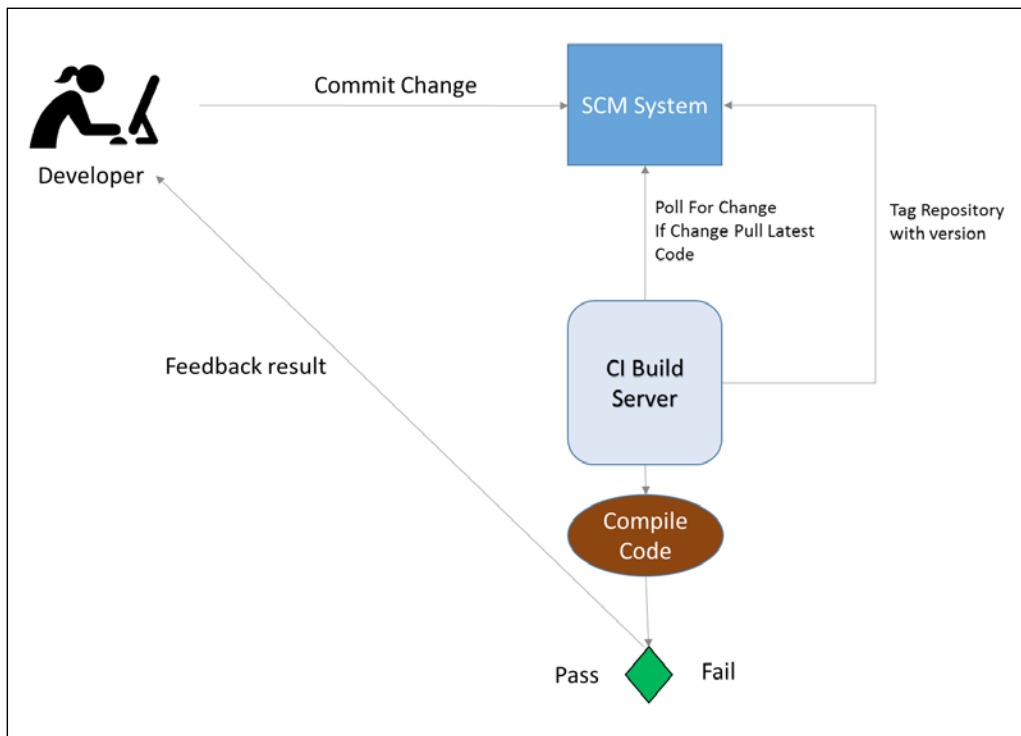
Although all IT staff may not follow identical deployment strategies, feedback loops and validation should not be unique to just developers. Sure, a compilation process may not be required when making network changes, but other validations can be done against a network device or a change on an SDN controller or load balancer to validate the changes are correct.

Developer continuous integration

A continuous integration process in its purest form takes a developer code change, integrates it with other developers' latest changes and makes sure it compiles correctly. The continuous integration process can then optionally run a set of unit or integration tests on the code base, package the compiled binaries, and then upload the build package to an artifact repository, tagging the code repository and package with a unique version number.

So, a simple continuous integration process can be summarized as the following feedback loop:

1. The developer commits code change to the **SCM System** and integrates it with the code base.
2. The code base is pulled down to a **CI Build Server**.
3. The code is compiled to check that the new commit is valid and non-breaking and the repository is tagged with the build version number.
4. Return **Pass** or **Fail** exit conditions and **Feedback result** to users.
5. Repeat steps 1-5 for the next code change.



Steps 1 (developer commit) and step 2 (creating a copy of the repository on the **CI Build Server**) are processes taken care of by **SCM Systems**.

Some of the popular SCM systems over the past 10 years have been Subversion, IBM Rational ClearCase, Microsoft Team Foundation Server, Perforce, and Telelogic CM Synergy. While distributed source control management systems have moved from centralized to distributed source control management systems such as Git and Mercurial in recent years.

Steps 3 (code compilation), 4 (code compilation feedback to users), and 5 (repetition of the process) in the process are carried out by a continuous integration building servers, which act as a scheduling agent for the continuous build process.

Tools such as Cruise Control, Hudson, or more recently Jenkins, Travis, and Thoughtworks Go are used to schedule continuous integration.

Step 4 (code compilation feedback to users) can be carried out using compilation tools such as:

- Maven <https://maven.apache.org/>
- Ant <http://ant.apache.org/>
- MsBuild [https://msdn.microsoft.com/en-us/library/ms171452\(v=vs.90\).aspx](https://msdn.microsoft.com/en-us/library/ms171452(v=vs.90).aspx)
- Rake <http://rake.rubyforge.org/>
- Make <http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/>

All these tools, and many more, can be used as the main validation step in the process depending on the type of code compilation that is required.

The continuous integration process is carried out, polling for every new developer commit, and carrying out the code compilation and repeating the same process over and over to provide a continuous feedback loop. If a developer breaks the CI build they need to immediately fix it so that it doesn't block other development changes from being compiled and validated. So developers use continuous integration to collaborate and make sure their changes successfully integrate.

Additional steps such as unit or integration tests can be subsequently bolted on to the process after the compilation is successful for increased validation of the change. Just because code compiles, it doesn't mean it is always functional. When all compilation and tests are packaged a sixth step may be introduced to package the software and deploy it to an artifact repository.

All good continuous integration processes should work on the premise of compile, test, and package. So a code release should be packaged once and the same package should be distributed to all servers at deployment time.

Database continuous integration

After continuous integration was set up to help improve the quality of code releases, developers that controlled database changes generally thought about doing similar processes for database changes. As database changes are always a big part of any enterprise release process having broken database releases can prevent software being deployed and released to customers.

As a result, database schema changes or database programmatic stored procedures would equally benefit from being integrated earlier on in the continuous integration process and tested in a similar way using quick validation and feedback loops.

In a way, developers have it easy when considering continuous integration as the compilation process is a binary pass or fail metric that is easy to understand. Scripting languages are of course the exception to this rule, but these can be supplemented using unit tests to provide the code validation on various code operations and both codes are improved by good test coverage.

When doing database schema changes, a number of test criteria need to be met prior to pushing the code to production. Good database developers will provide roll forward and roll-back scripts when making SQL changes, which will be applied to production databases and they normally test these on their development machines prior to checking them into a source control management system.

Database developers implement database changes using a roll-forward and roll-back release script and store them in SCM systems. The roll-back is only performed in the case of an emergency when it is being applied to production if the roll-forward for any reason fails.

So a typical database release process will have the two following steps:

- Apply SQL table or column creation, update, deletion, or stored procedure using release script.
- If this fails, roll-back SQL table or column creation, update, deletion or stored procedure using the roll-back release script.

So prior to any production release, a database developer's roll-forward and roll-back scripts should be tested. As multiple database developers are part of the same release, these database release scripts should be applied in the same sequenced order as they would be applied to production as one developers change could break another developers changes.

Before setting up a database continuous integration, a few prerequisites are required:

- A database schema matching production with a relative dataset and all the same characteristics such as indexing so we are testing against a similar live version
- The continuous integration process should also utilize the same deployment runner script that is used to sequence the database release scripts and provide roll-back in case of failure

Testing roll-back scripts is as integral to testing roll-forward scripts so the database continuous integration process will need valid tests to encompass roll-back.

A common database deployment workflow applied by a database developer on their local workstation would look like this:

1. Apply roll-forward database script using deployment runner script to CI test database.
2. Apply roll-back database script using deployment runner script to CI test database.
3. Apply roll-forward database script using deployment runner script to CI test database.
4. Apply roll-back database script using deployment runner script to CI test database.

If the preceding set of steps is successful then the roll-forward and roll-back database scripts are sound in terms of syntax and won't fail when applied to the production database.

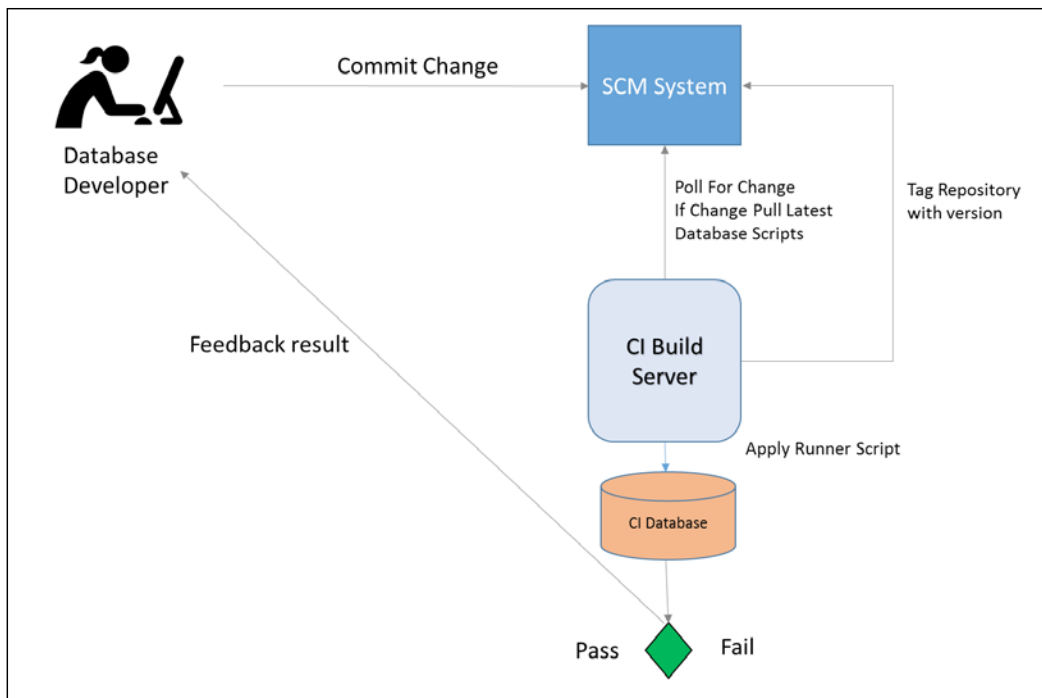
The preceding steps also check the validity of the sequencing using the deployment runner and check that the integrated database deployment scripts work together and do not conflict on roll-back either.

Using continuous integration, we have already ruled out multiple possible scenarios that could cause a failure in production. However, the preceding continuous integration process alone is not enough, as with a code compilation, just because SQL is not returning an error doesn't mean the database roll-forward and roll-back scripts are technically valid, so database changes still need to be supplemented with functional tests too.

Continuous integration is about putting quality checks earlier in the delivery lifecycle and creating feedback loops. Continuous integration is not about proving that a release is 100% valid, it should instead be looked at as a way of proving that a checking process has been followed, which proves that a release is not broken.

A simple continuous integration database process would provide the following feedback loop for database developers:

1. Developer commits roll-forward and roll-back change to **SCM System** and it is integrated with the code base.
2. The code base is pulled down to a **CI Build Server**.
3. Apply roll-forward database script using deployment runner script to CI test database.
4. Apply roll-back database script using deployment runner script to CI test database.
5. Apply roll-forward database script using deployment runner script to CI test database.
6. Apply roll-back database script using deployment runner script to CI test database.
7. Return **Pass** or **Fail** exit condition and feedback to users.
8. Repeat steps 1-7 for the next database change.



Once the release is ready to go live the database CI will have the final changes applied, preparing it for the next release, the next iteration of database changes and the next batch of database scripts that will be applied by the next release. Alternately, the CI database schema can be refreshed from production.

A good concept is to always create a baseline of the database so that if a database developer unwittingly commits a bad roll-forward and roll-back on a database then the CI database can be easily restored to the desired state and not prove a bottleneck for development.

Of course this is one way of dealing with validation of database changes and others are possible. Microsoft offers database projects for this very purpose, but the validation engine is not important, having validation of any changes early in the release lifecycle is the important takeaway.

It is important to make sure that nothing goes to production unless it goes through the CI process, there is no point setting up a great process and then skipping it as it makes the CI database schema invalid and could have massive consequences.

Tooling available for continuous integration

Many different flavors of configuration management tooling are available to help build continuous integration processes, so there is a rich variety of different options to choose from, which can seem daunting at first.

Tools should be picked to facilitate processes and will be selected by teams or users. As described in *Chapter 3, Bringing DevOps to Network Operations*, it is important to first map out requirements that need to be solved and the desired process before selecting any tooling.

By the same token, it is important to avoid tools sprawl, which is all too common in large companies and have only one best fit tool for every operation rather than multiple tools doing the same thing as there is an operational overhead for the business.

If configuration management tooling already exists in a company for continuous integration then it will more than likely be able to meet the needs. When considering the tooling for carrying out continuous integration processes the following tools are required:

- SCM system
- Validation engine

The SCM system is primarily used for storing code or configuration management configuration in a source control repository.

The validation engine is used to schedule the compilation of code or validate configuration. So continuous integration build servers are used for the scheduling and numerous compilation or test tools can be used to provide the validation.

Source control management systems

SCM systems provide the center of a continuous integration process, but no matter the SCM system that is chosen; at a base level it should have the following essential features:

- Be accessible to all users that need to push changes
- Store the latest version of files
- Have a centralized URL that can be browsed by users to see available repositories

- Have a role-based access permission model
- Support roll-back of versions and version trees on files
- Show which user committed a change along with the date and time of the change
- Support tagging of repositories, this can be used to check out a tag to show all the files that contributed to a release
- Support multiple repository branches for parallel development
- Have the ability to merge files and deal with merge conflicts
- Have a command line
- Plug into Continuous Integration Build servers

Most SCM systems will also support additional features such as:

- A programmable API or SDK
- Easily integrated with developer IDEs
- Integrate with Active Directory or **Lightweight Directory Access Protocol (LDAP)** for role-based access
- Support integration with change management tools, where a SCM commit can be associated with a change ticket
- Support integration with peer review tools

SCM systems can either be centralized or distributed, in recent years, distributed source control management systems have increased in popularity.

Centralized SCM systems

When SCM systems were originally created to facilitate development teams, a centralized architecture was used to build these systems. A centralized SCM system would be used to store code and developers would access the repository they were required to make code changes against and make edits against a live centralized system.

For developers to remain productive, the centralized SCM system would always need to be available and online:

- Developers would access the repository where they wish to make code changes
- They'd then check out the file they wished to edit
- Make changes
- Then check the file back into the code central branch

The SCM system would have a locking mechanism to avoid collisions where only one file can be edited by one user at a time. If two developers accessed the file at the same time, the online SCM system would say it was locked by another developer and they would have to wait until the other developer made their change prior to being allowed to check out the code and make the subsequent change.

Developers when making changes would make a direct connection to repositories hosted in the centralized SCM system to make code updates. When a developer made a change, this in turn would write the changes in state to a centralized database, updating the state of the overall repository.

The state change would then be synchronized to other developer's views automatically. One of the criticisms of centralized SCM systems was the fact that developers sometimes wanted to work offline, so some centralized source control management systems introduced the concept of snapshot views, which was an alternative to the permanently live and updated repository view and also introduced offline update features.

A snapshot view in a centralized SCM system was a snapped copy of the live repository at a given point in time. Best practice would dictate that before committing any development changes to the centralized server, the snapshot view should be updated; any merge conflicts would be dealt with locally before checking in any changes that were made in the snapshot view.

Developers would integrate with the centralized SCM system using the command-line interface or GUI that was integrated with a developers IDE for ease of use so they didn't need to jump between the command line and the IDE.

Examples of good centralized source control management systems are as follows:

- IBM Rational ClearCase
- Telelogic CM Synergy
- IBM Rational Team Concert
- Microsoft Team Foundation Server
- Subversion
- Perforce

Distributed SCM systems

Distributed SCM systems do not have a central master and instead replicate changes to multiple places. Users will create replicas of a repository and then can pull or push using their own local copy sitting on their local development machine.

Each repository in a distributed system will have an owner or maintainer and users will submit changes in the form of pull requests. Developers will create a pull request, which is like a merge request, but instead the repository maintainer can then approve if they accept the pull request or not. Once accepted, the commit will be pulled into the branch.

One of the main benefits of a distributed SCM system is the ability to work on the repository offline. Changes can be committed to the local repository and then once it's back online, pushed to the master branch when developers are ready.

Distributed SCM systems are more merge-friendly and efficient, so they work better with agile development, which often means multiple small repositories for each microservice rather than large centralized code bases for monolith applications.

Examples of distributed SCM systems are as follows:

- Git
- Mercurial
- Veracity

Branching strategies

Branching strategies are used to meet the needs of modern software development, with multiple branches serving different use cases and supporting multiple versions of the code.

SCM systems traditionally relied on a **Mainline** branch, often referred to as the **Trunk** or **Master** branch. A mainline branching strategy meant that the mainline/trunk branch is always the clean and working version of the code, and the files on this branch are representative of the code in production.

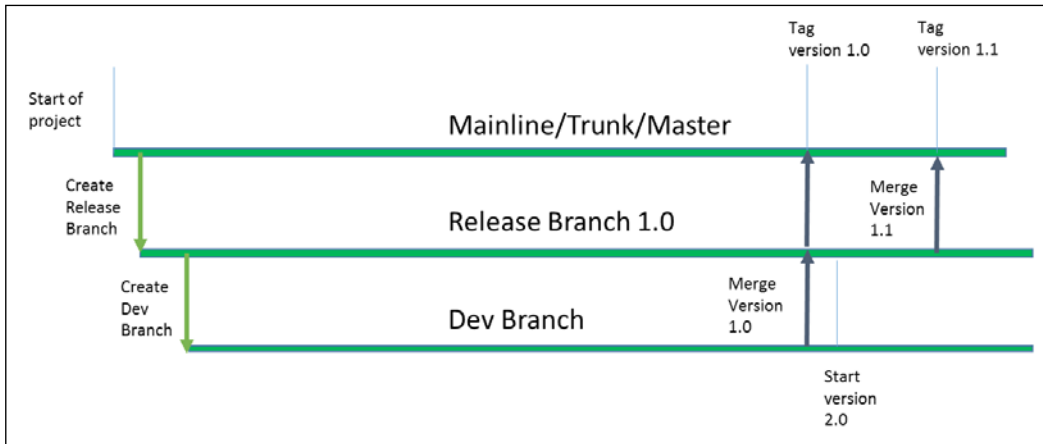
Development branches were then created for active development on the latest releases, while release branches were used for maintenance releases if bugs were identified on the production system.

There are many different branching strategies that can be implemented; in the following example, the mainline branching strategy is illustrated.

The mainline/trunk/master branch is kept clean and all releases are done by merging changes to it, and this branch is tagged every time a release is done. This allows a diff to be done between tags to see what has changed.

The development branch is used for active development and creates version 1.0, then merges to the **Release Branch 1.0**, which in turn immediately merges back to **Mainline/Trunk/Master**.

The development branch then starts active development on version 2.0, while **Release Branch 1.0** is used for 1.x maintenance releases if a bug fix is required:



The mainline branching strategy meant a lot of merging and coordination and release managers were required to coordinate merges and releases of versions on release days.

Centralized configuration management systems were set up to favor a mainline approach to software development and this was good when supporting waterfall development.

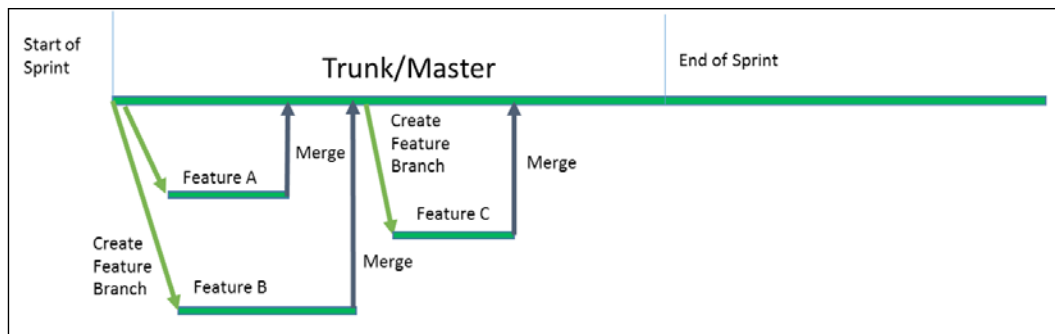
Waterfall software development has rigid phases of the project, incorporating analysis, design, implementation, and testing phases, so the mainline branching strategy was sufficient when teams were producing only one release every few months as opposed to daily releases, so the laborious merge process was not such a bottleneck.

However, the transition to agile software development meant that implementing the mainline strategy became more difficult as teams release more frequently now that they have moved towards continuous deployment and delivery models.

An alternate branching strategy better suited to agile development is using feature branches. In agile software development work is split into sprints that last two weeks. So the master or mainline branch is still used but very short-lived feature branches are created by developers during a sprint. Distributed SCM systems put the developer in charge of the merging as opposed to using a centralized release management team for these operations.

In the following example, we can see an example of feature branching, where three different feature branches, **Feature A**, **Feature B**, and **Feature C** are created during a two week sprint. When developers have finished development their features merged back into the **Trunk/Master** branch.

Every time a commit is done from a feature branch then the change is merged directly to **Trunk/Master** and a continuous integration process will be started which will validate the changes, every successful check-in then becomes a potential release candidate. After a release is packaged by the continuous integration process it is ready for deployment, as shown in the following diagram:



Some purists will argue against utilizing feature branches at all, preferring to always work against **Trunk/Master**. However, this decision is down to the individual teams to govern which approach works best for them and it is subjective. Some will also argue that it adds an additional level of control until adequate testing is created on the **Trunk/Master** branch so that changes can be suitably peer-reviewed prior to merging.

When a commit is done against a branch it should trigger a CI build and associated validation of whatever change has been committed. This creates feedback loops at every stage of the process. Any change that goes into any branch should be governed by a CI build to gate-keep good changes and highlight breaking changes as soon as they happen so they can be fixed immediately.

Continuous integration build servers

Various continuous integration build servers are available to help schedule validation steps or tests. One of the first continuous integration build servers was Cruise Control from Thoughtworks, which has since evolved into Thoughtworks Go.

Cruise Control allowed users to configure an XML file that set up different continuous integration build jobs. Each build job ran a set of command-line options; normally, a compilation process against a code repository and it returned a green build if it was successful and a red build if the build was broken. Cruise Control would highlight the errors in the form of build logs providing feedback to users via the Cruise Control dashboard or by e-mail.

The market leading build server at the moment is Cloudbees Jenkins, which is an open source project and a fork of the original Hudson project. Jenkins really took away the need to configure XML files and moved all setup operations into the GUI or API. It comes with a plethora of plugins that can pretty much carry out any continuous integration operation possible. It also has recently delved into Continuous Delivery as of Jenkins 2.x release.

The next evolution of CI systems has moved towards cloud-based solutions with Travis being a popular choice for open source projects. This allows users to check in a Travis YAML file, which creates the build configuration from source control and can be versioned along with the code. This is something Jenkins 2.x is doing now using the Jenkinsfile and that the Jenkins job builder project had been doing for the OpenStack project.

There are many different options when looking for continuous integration build servers, consider the following; no matter the continuous integration build system that is chosen, at a base level it should have the following essential features:

- Dashboard for feedback
- Notion of green and red builds
- Scheduling capability for generic command lines
- Pass or fail builds based on exit conditions, 0 being a pass
- Plug-ins to well-known compilation tools
- Ability to poll SCM systems
- Ability to integrate with unit testing framework solutions such as Junit, Nunit, and more
- Role-based access control
- Ability to display change lists of the latest commits to a repository that has been built

Most continuous integration build servers will also support additional features such as:

- Have a programmable API or SDK
- Provide e-mail or messaging integration

- Integrate with Active Directory or LDAP for role-based access
- Support integration with change management tools, where a SCM commit can be associated with a change ticket
- Support integration with peer review tools

Network continuous integration

So why should network engineers be interested in continuous integration?

A network team should be interested in continuous integration if they want to improve the following points, which were focused on in *Chapter 3, Bringing DevOps to Network Operations*:

- Velocity of change
- Mean time to resolve
- Improved uptime
- Increased number of deployments
- Cross skilling between teams
- Removal of the bus factor of one

The ability to easily trace what has changed on the network and see which engineer made a change is something that continuous integration brings to the table. This information will be available by looking at the latest commit on a continuous integration build system.

Roll-back will be as simple as deploying the last tagged release configuration as opposed to trawling through device logs to see what changes were applied to a network device if an error occurs.

Every network engineer can look at the job configuration on the continuous integration build system and see how it operates so every network engineer knows how the process works so it helps with cross skilling.

Having continual feedback loops will allow network teams to continuously improve processes, if a network process is sub-optimal then the network team can easily highlight the pain points in the process and fix them as the change process is evident to all engineers and done in a consistent manner.

When network teams use continuous integration processes it moves network teams out of firefighting mode and into tactical continuous improvement and optimization mode. Continuous integration means that the quality of network changes will improve as every network change has associated validation steps that are no longer manual and error prone.

Instead, these checks and validations are built in and carried out every time a network operator commits a network change to the SCM System. These changes can be built up over time to make network changes less error prone and give network engineers the same capabilities as developers and infrastructure teams.

Utilizing network continuous integration also takes the fear out of making production changes, as they are already validated and verified as part of the continuous integration process, so production changes can be viewed as just a business-as-usual activity, rather than something that needs to be planned weeks in advance or worried about. The view is: if an activity is problematic then do it more often, continually iterate it, improve it, and make people less afraid of doing it.

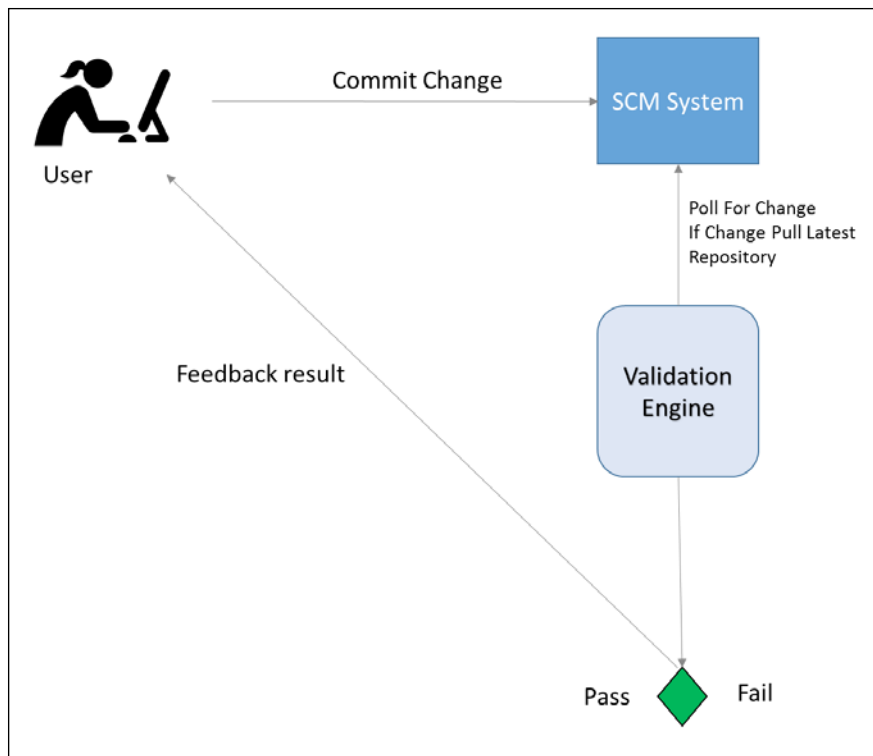
Having covered topics such as different SCM branching strategies, continuous integration build servers, and shown how continuous integration can be used for code and database changes, it should now be clear what continuous integration is and that it is not just about compilation of code. Instead, continuous integration is about validating parallel changes, making sure they all work together and providing feedback loops to users.

The DevOps movement is about interacting with others and removing bottlenecks, delivering products to market faster, and increasing accuracy so continuous integration is equally applicable to networking. The automation of processes and the collaboration between teams using similar concepts is very important so continuous integration really is the glue that holds infrastructure and networking as code together.

To a network engineer, concepts such as continuous integration may seem alien at first, but instead of talking about deep dive compilation processes, it should be focusing on processes. If any network engineer was asked if they could have a quick and easy-to-use process that validated all their network changes before production, providing quick feedback loops, then the answer would be yes. Continuous integration can therefore be a useful tool that would mean less broken production changes.

In this book, in *Chapter 4, Configuring Network Devices Using Ansible*, *Chapter 5, Orchestrating Load Balancers Using Ansible*, and *Chapter 6, Orchestrating SDN Controllers Using Ansible*, we looked at ways that network changes could be treated as code, using configuration management tooling such as Ansible to configure network devices, load balancers, and SDN controllers.

So when considering the following diagram, the question regarding continuous integration of network changes is not asking if continuous integration is possible for network changes. It should instead be questioning which validation engines can be used for network changes after a SCM commit has taken place to give a quick feedback loop of **Pass** or **Fail** to network operators:



Network validation engines

The challenge when creating continuous integration builds for network changes is what to use for the validation engine. Network changes when using Ansible rely heavily on YAML configuration files, so the first validation that can be done is checking the YAML `var` files.

The `var` files are used to describe the desired state of the network, so checking that these YAML files are valid in terms of syntax is one valid check. So to do this, a tool such as `yamllint` can be used to check if the syntax of the files that are committed into source control management are valid.

Once the YAML `var` files are checked into source control, the continuous integration build should create a tag to state a new release has happened. All SCM systems should have a tagging or base-lining feature.

Tagging versions means that the current network release version can be diffed against the previous version to see what file changes have occurred on the YAML `var` files. If an issue is detected at any stage, all networks changes are made transparent.

So what other validation is possible? When focusing on configuration of network devices, we are pushing configuration changes to a networking operating system such as Juniper Junos or Arista Eos. So being able to run the newly committed changes and make sure the syntax is programmatically correct against those operating systems as part of the continuous integration process is highly desirable. Most network device operating systems as discussed in *Chapter 4, Configuring Network Devices Using Ansible*, are Linux-based, so having a network operating system to issue commands to as part of the CI process doesn't seem too absurd.

The same can be said when checking the configuration used to orchestrate load balancers or SDN controllers, having a test environment attached to the continuous integration process is also highly desirable in theory. By utilizing a software version of the load balancer or emulated version of the SDN controller would be highly beneficial, so network engineers can pre-flight their network changes to make sure the API calls and syntax is correct.

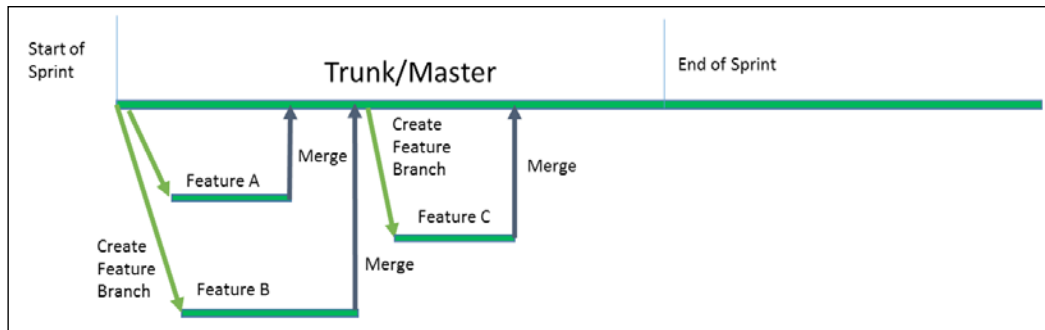
However, there are challenges with simulating an SDN controller or creating or simulating a production environment depending on the vendor, they may have a huge overhead in terms of setting up a continuous integration environment due to cost. Network devices, load balancers, and SDN vendors are evolving to support automation and DevOps friendly processes such as continuous integration. Therefore, networking vendors are starting to appreciate the validity of giving small test environments; this is where virtualized or containerized versions of load balancers or SDN controllers would be useful as an API endpoint to validate the desired state that has been set up in YAML files.

Alternately, the vendors could provide a vagrant box to test if the desired configuration specified in YAML `var` files that is checked into SCM Systems is valid before it is propagated to the first test environment. Any enhancements that can be done to processes to make it fail as fast as possible and shift issues as far left as possible in the development lifecycle should be implemented where possible.

So with all of these validators, let's look at how these processes can be applied to network devices, or alternately orchestration. The number of validators used may depend on the network vendors that are being used, so we will look at the start point for a continuous integration build for network devices regardless of vendor and then look at more advanced options that could be used if the vendor provides a software load balancer or SDN emulation.

Simple continuous integration builds for network devices

As network changes are always required daily by large organizations that are implementing microservice applications. To meet those demands networking should be as self-service as possible. To keep up with demand, network teams will probably need to use a feature branch SCM strategy or allow self-service YAML files to be committed directly to the master branch, as shown in the following diagram:



Each commit should be peer reviewed before it is merged. Ideally, the self-service process should allow development teams to package network changes alongside their code changes and follow a self-service approach.

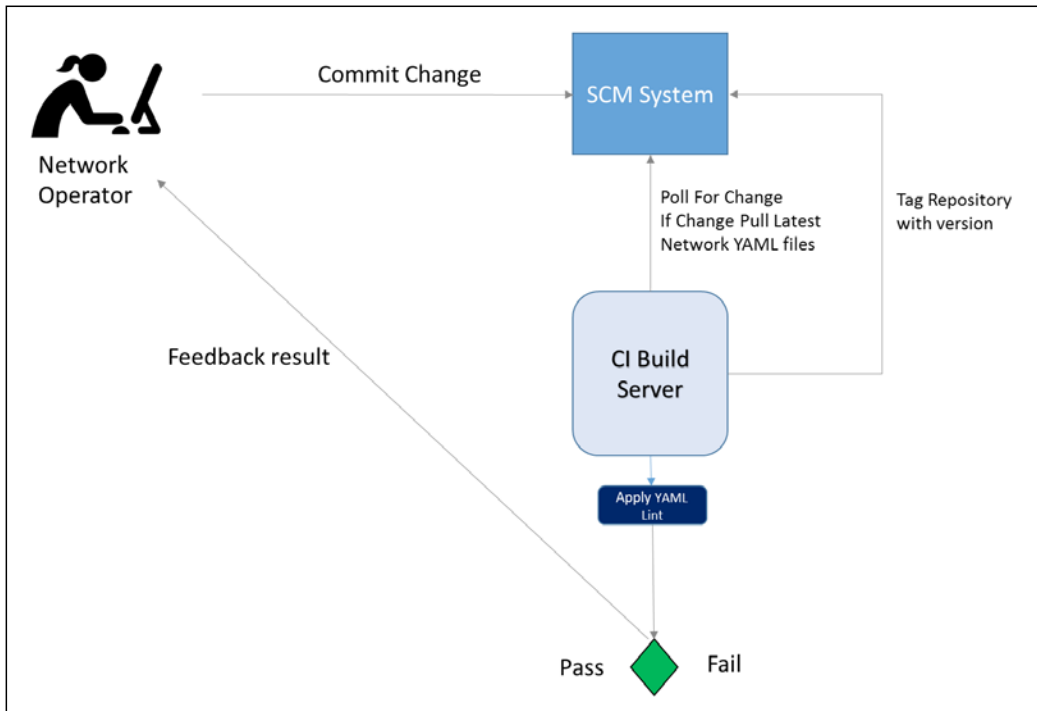
The first continuous integration build that should be set up for network devices or orchestration should focus on version controlling the Ansible YAML files and running a simple YAML validation on the desired state.

Each continuous integration build that runs will also tag the repository. Tagging the SCM repository means that release versions can be compared or easily rolled back. It will also act as an audit log to show which user made changes and what exactly has changed in the environment. No changes should be made to a production system that has not gone through the continuous integration process.

So a simple network continuous integration build will follow these simple validation steps:

1. YAML files are checked for syntax.
2. The repository is tagged in the SCM System if successful.

Therefore, a simple network continuous integration build would follow these steps. The network operator would commit the YAML files to the SCM system to change the desired state of the network; the continuous integration build server would tag the build if the YAML Lint operation finds that all the YAML files in the repository have valid syntax and return a positive result:

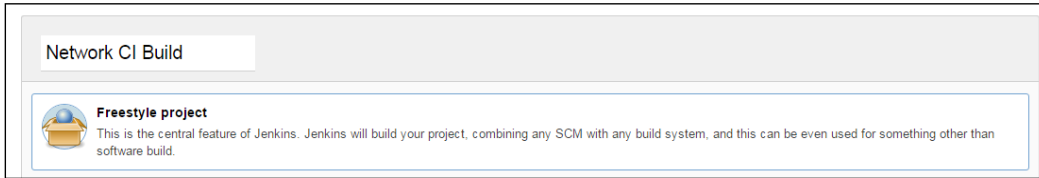


Configuring a simple Jenkins network CI build

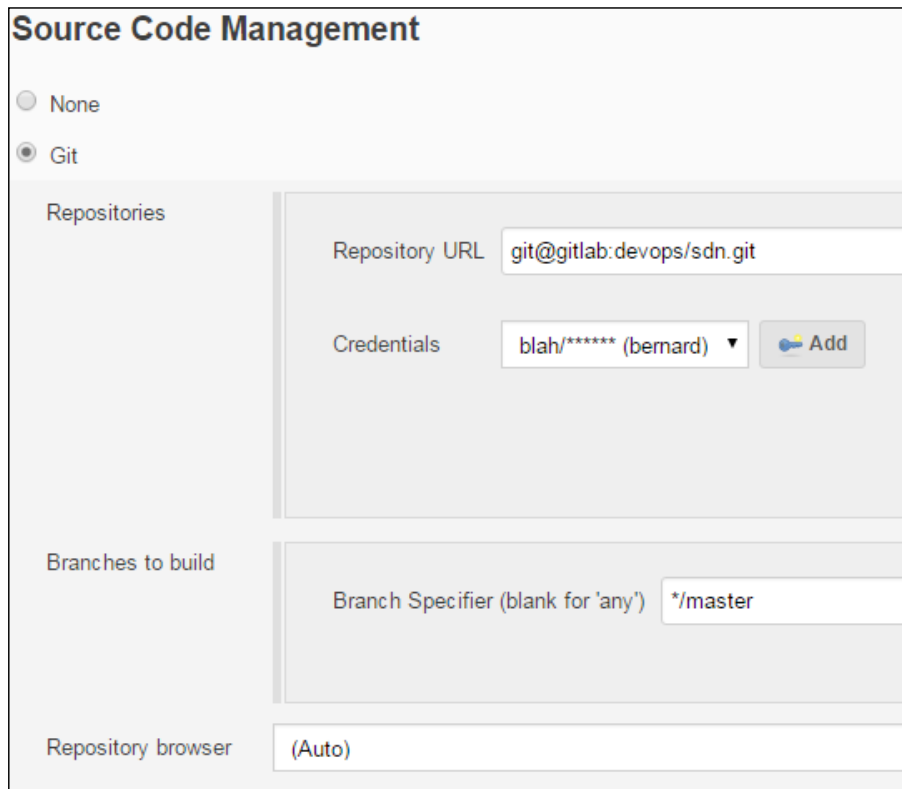
This simple continuous integration build for network devices can be set up in the Jenkins CI build server. Rake and the `yamllint` gem should be configured on the Jenkins slave that the build will be executed on.

Once this has been completed, a new Jenkins CI build can be created in a matter of minutes.

First, select a new Jenkins freestyle job:

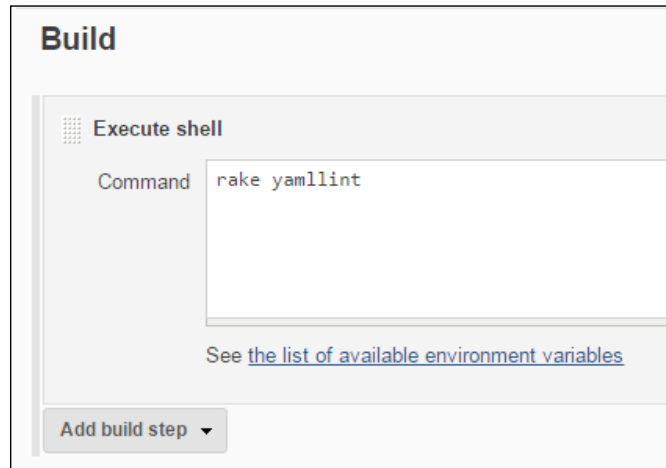


Then configure the SCM system to use, in this instance Git, specifying `git@gitlab:devops/sdn.git` as the repository and the `*/master` branch of the project along with the SSH key required to provide access to the repository:

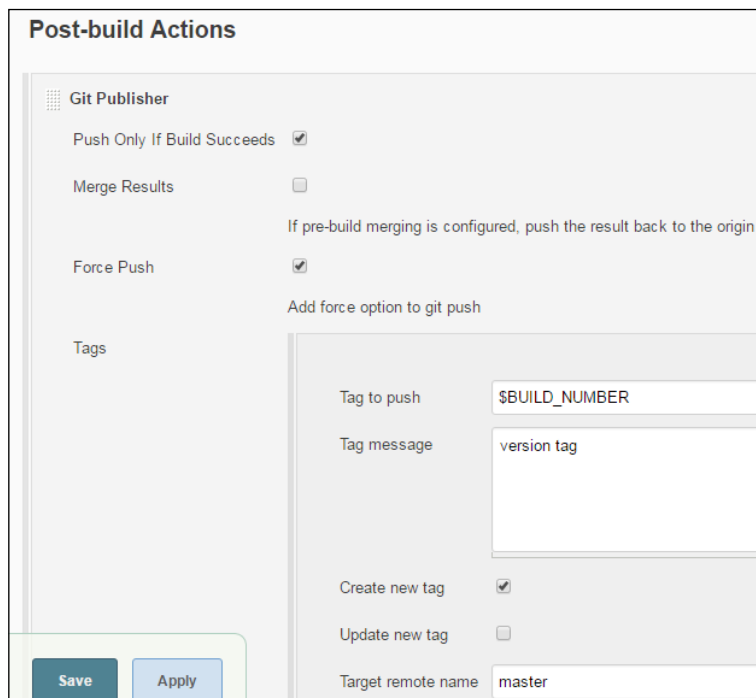
A screenshot of the 'Source Code Management' configuration page in Jenkins. The page has a title 'Source Code Management' and two radio buttons: 'None' and 'Git'. The 'Git' radio button is selected. Below the radio buttons, there are three sections: 'Repositories', 'Branches to build', and 'Repository browser'. In the 'Repositories' section, the 'Repository URL' is set to 'git@gitlab:devops/sdn.git' and the 'Credentials' dropdown is set to 'blah/***** (bernard)' with an 'Add' button next to it. In the 'Branches to build' section, the 'Branch Specifier (blank for 'any')' is set to '*/master'. In the 'Repository browser' section, the value is '(Auto)'.

Source Code Management	
<input type="radio"/> None	
<input checked="" type="radio"/> Git	
Repositories	Repository URL: git@gitlab:devops/sdn.git
	Credentials: blah/***** (bernard) [Add]
Branches to build	Branch Specifier (blank for 'any'): */master
Repository browser	(Auto)

Now for the validation step, a shell command build step is selected, which will run `rake yamllint` on the repository after configuring a **Rakefile** in the `git@gitlab:devops/sdn.git` repository so the YAML files can be parsed:

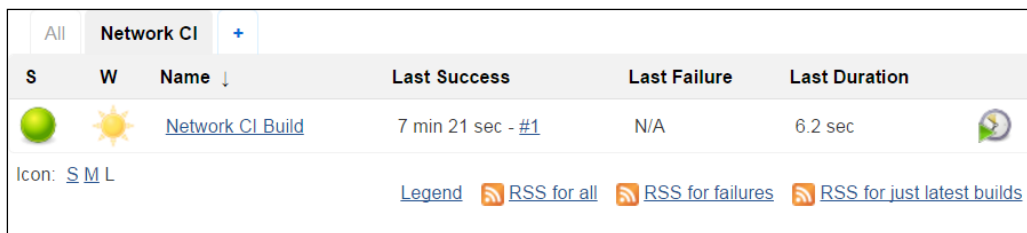




Finally, configure the build job to tag the Jenkins build version against the `devops/sdn.git` gitlab repository and **Save** the build:






This has configured a very simple Jenkins CI build process that will poll the Git repository for new changes, run `yamllint` against the repository, and then tag the Git repository if the build is successful.

The build health will be shown in Jenkins; the green ball means the build is in a healthy state so the YAML files are currently in a good state, and the duration of the check shows it took 6.2 seconds to execute the build, as shown in the following screenshot:



S	W	Name ↓	Last Success	Last Failure	Last Duration
		Network CI Build	7 min 21 sec - #1	N/A	6.2 sec

Icon: [S](#) [M](#) [L](#)

[Legend](#)  [RSS for all](#)  [RSS for failures](#)  [RSS for just latest builds](#)

Adding validations to network continuous integration builds

After highlighting the need for more robust validation to pre-flight configuration of network devices shifting failure as far left in the development lifecycle as possible to reduce the cost to fix. Having the ability to push mission-critical configuration changes to a networking operating system such as Cisco Nxos, Juniper Junos, or Arista Eos would be a good continuous integration validation.

So, like databases verifying that SQL syntax is correct, being able to run the newly committed changes and make sure the networking commands or orchestration commands applied to network devices syntax is programmatically correct should be part of the continuous integration build.

Continuous integration can then help the quality of network changes as an incorrect change would never be pushed to a network device, load balancer, or SDN controller. Of course, the functionality of the configuration pushed may not be what is required, but there should at least never be a situation where the configuration has a syntax error at deployment time.

As network devices, load balancer, and SDN controller changes are mission-critical, this brings an added layer of validation checks to any network changes and checks in a quick and automated way, providing quick feedback if a network change is not what is required.

Continuous integration for network devices

Before setting up a network device, continuous integration of a few prerequisites is required:

- A network operating system will be required with production configuration pushed to it and all live settings, which can be hosted on a virtual appliance
- The continuous integration build tools such as Jenkins will need to have an Ansible Control Host set up on the agent so it can execute Ansible playbooks
- All playbooks should be written with a block rescue so subsequent cleanup is built-in if the execution of the playbook fails

A typical network device release process will have the two following steps:

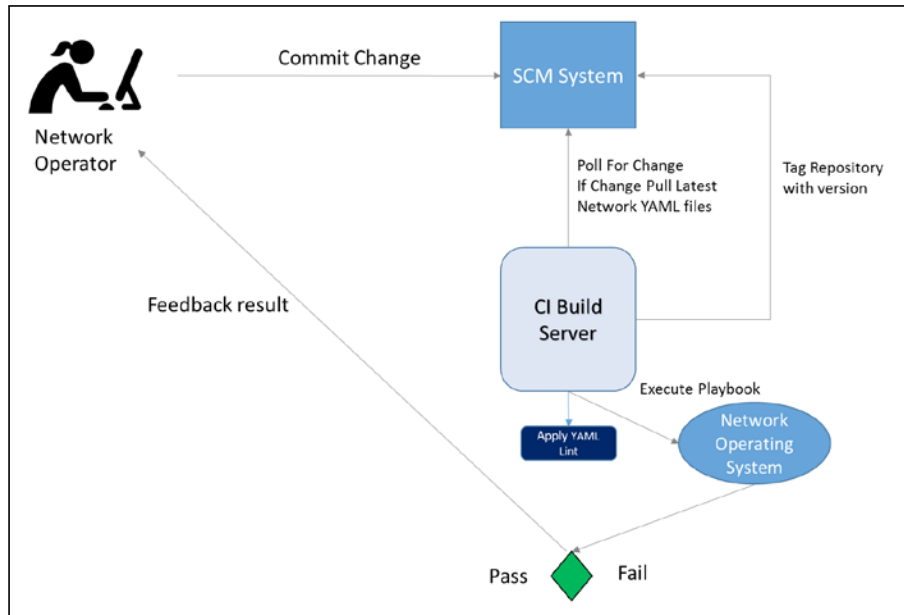
1. Apply the network change self-service playbook.
2. As the playbook is idempotent, changes will only be shown if a change has occurred.

The Ansible playbook should provide resilience for roll-forward and roll-back in terms of state change. The previous steps also check the validity of the sequencing using the Ansible playbook and also check that the calls being made to the network device are valid.

A simple continuous integration network build process would provide the following feedback loop for network operators:

1. The network operator commits Ansible playbook or YAML `var` file's change to **SCM System**, and it is integrated with the code base.
2. The code base is pulled down to a **CI Build Server**.
3. YAML files are checked using `yamllint`.
4. The Ansible playbook is applied to push network changes to the device.
5. Return **Pass** or **Fail** exit conditions and feedback to users.

6. Repeat steps 1-5 for the next network device change:



Continuous integration builds for network orchestration

Before setting up a network orchestration for load balancers or SDN controllers, a few prerequisites are required:

- A software load balancer or an emulated SDN controller will be required with production configuration pushed to it and all live settings
- The continuous integration build tools such as Jenkins will need to have an Ansible controller set up on the agent so it can execute Ansible playbooks as well as the SDK that will allow the network orchestration modules to be executed
- All playbooks should be written with a block rescue so subsequent cleanup is built in if the execution of the playbook fails

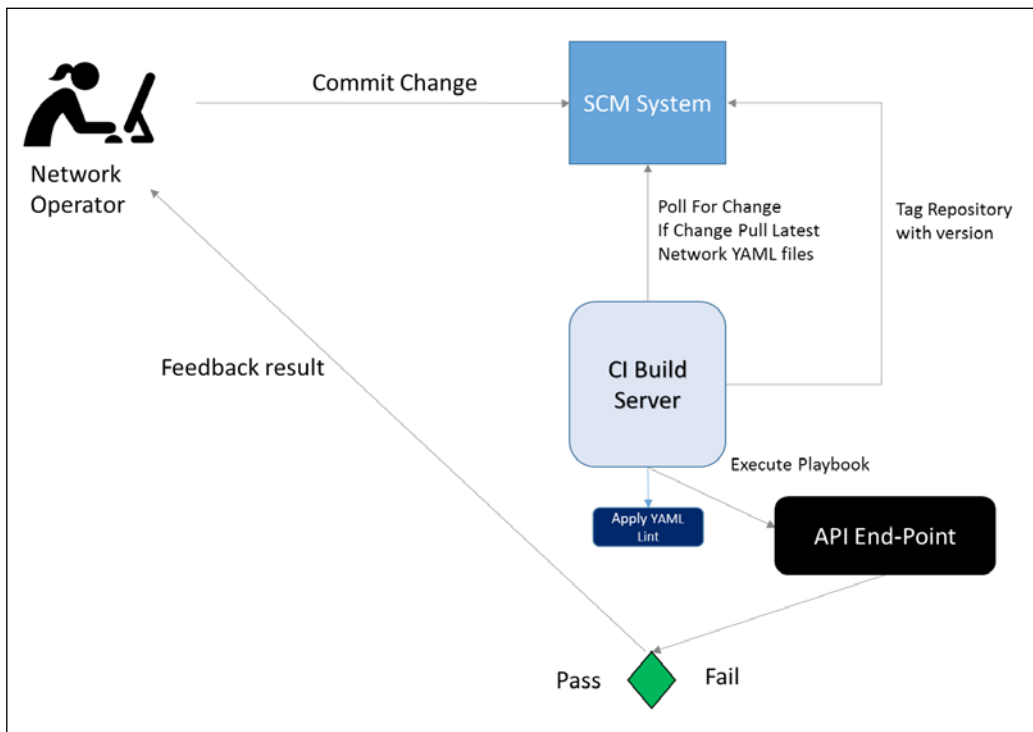
A typical network device release process will have the following steps:

- Apply network changes to the self-service playbook.
- As the playbook is idempotent, changes will be only shown if a change has occurred.

The Ansible playbook, like with the network device changes should provide resilience for roll-forward and roll-back in terms of state change. Some test servers may be needed on a virtualization platform to simulate the load balancing so health checks can be tested too.

A simple continuous integration network orchestration continuous integration process would provide the following feedback loop for network operators:

1. The network operator commits Ansible playbook or YAML var file change to **SCM System** and it is integrated with the code base.
2. The code base is pulled down to a **CI Build Server**.
3. YAML files are checked using `yamllint`.
4. An Ansible playbook is applied to orchestrate the API and create the necessary load balancer or SDN changes.
5. Return **Pass** or **Fail** exit condition and feedback to users.
6. Repeat steps 1-5 for next network orchestration change:



Summary

In this chapter, we have looked at what continuous integration is and how continuous integration processes can be applied to code and databases. The chapter then looked at ways that continuous integration can be applied to assist with network operations to provide feedback loops.

We also explored different SCM methodologies, the difference between centralized and distributed SCM systems and how branching strategies are used with waterfall and agile processes.

We then looked into the vast array of tools available for creating continuous integration processes focusing on some examples using Jenkins to set up a simple network continuous integration build.

In this chapter, you learned what continuous integration is, how it can be applied to network operations, SCM tooling, and the difference between centralized and distributed systems along with common SCM branching strategies.

Other key takeaways from this chapter include continuous integration build servers and their use, ways to integrate network changes into continuous integration, and potential continuous integration validation engines for network changes.

In the next chapter, we will look at various test tools and how they can be applied to continuous integration processes for added validation. This will allow unit tests to be created for network operations to make sure the desired state is actually implemented on devices before we will look at deploying the network changes in Continuous Delivery pipelines.

8

Testing Network Changes

This chapter will focus on an important part of the software development lifecycle as well as DevOps, testing, and quality assurance. This chapter will describe why it is essential to incorporate network changes as part of the continuous integration process and test them thoroughly. It will then go on to look at open source test tooling that is available to facilitate the creation of tests suites for network operations.

This chapter will focus on the overall quality assurance process, outlining some of the best-practice approaches that can be adopted by network teams or teams implementing network operations.

We will also look at the benefits of implementing feedback loops, quality reporting, and what checks can be implemented to make sure that the network is functioning as expected. These are all essential topics as network teams move toward code-driven network operations.

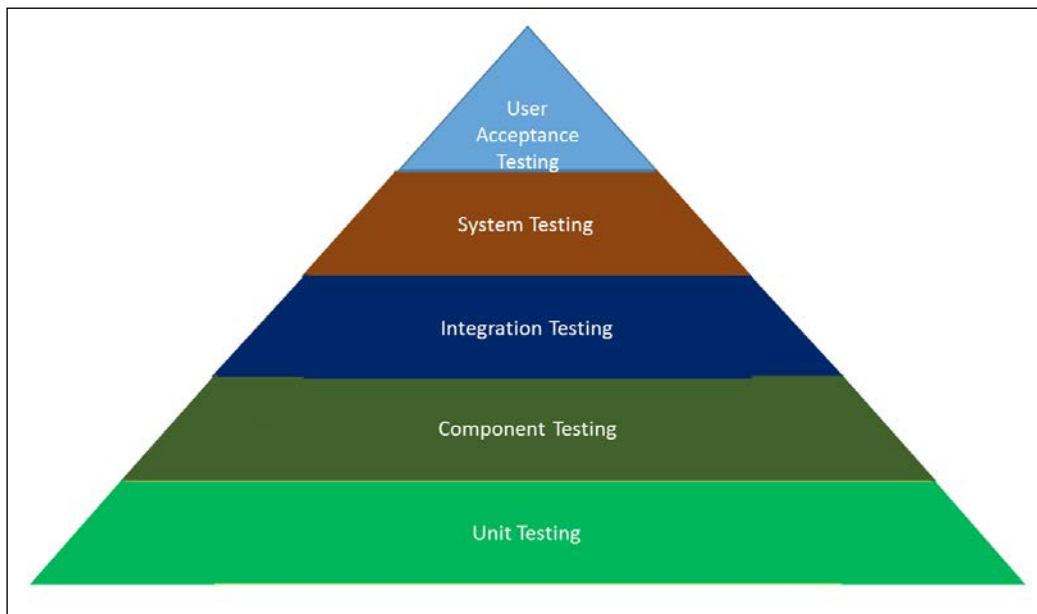
In this chapter, the following topics will be covered:

- Testing overview
- Quality assurance best practices
- Available test tools

Testing overview

There are many ways to ensure quality when making operational or development changes.

When combining quality checks and ordering them, they can be used to form a set of quality gates that development, infrastructure, or even network changes should flow through before they reach production. We will briefly touch upon some of the more popular testing strategies that are used to ensure that any changes to a system or application are operating effectively, comprised of the following phases of testing:



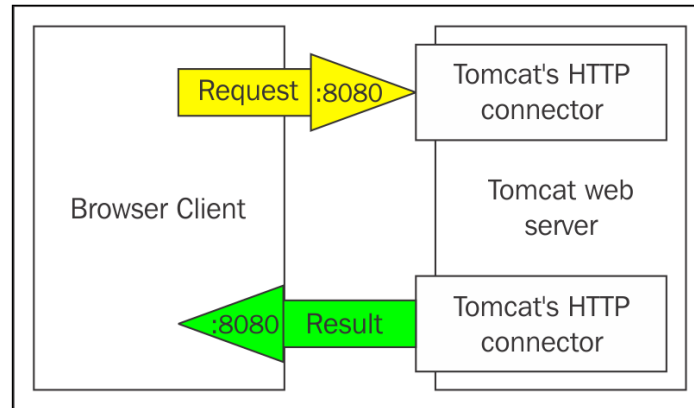
Unit testing

One of the most popular types of quality assurance is the unit test. A **unit test** will test each isolated code operation and make sure that each method or function exhibits the desired behavior with different inputs.

One or more unit tests will be required to make sure that a method or function works as desired. So multiple unit tests may need to be written to test any basic operation asserting either a pass or failure based on one isolated operation.

Unit tests can normally be carried out against compiled binaries, as opposed to requiring a fully-fledged test environment. Utilizing popular test frameworks, unit tests can be used to assert a pass or failure based on input.

For example, a unit test for an Apache Tomcat web server could involve making sure that the code can serve traffic on HTTP port 8080:



Component testing

Component testing involves testing a single component in isolation and making sure that it behaves the way it should as a self-contained entity.

Component testing normally involves deploying an application to a test environment and executing a suite of tests against the component that tests all its features and functionality. Microservice applications are small components which need to be tested each time they are released.

This may involve making sure a banking application can process transactions correctly based on a specific type of account.

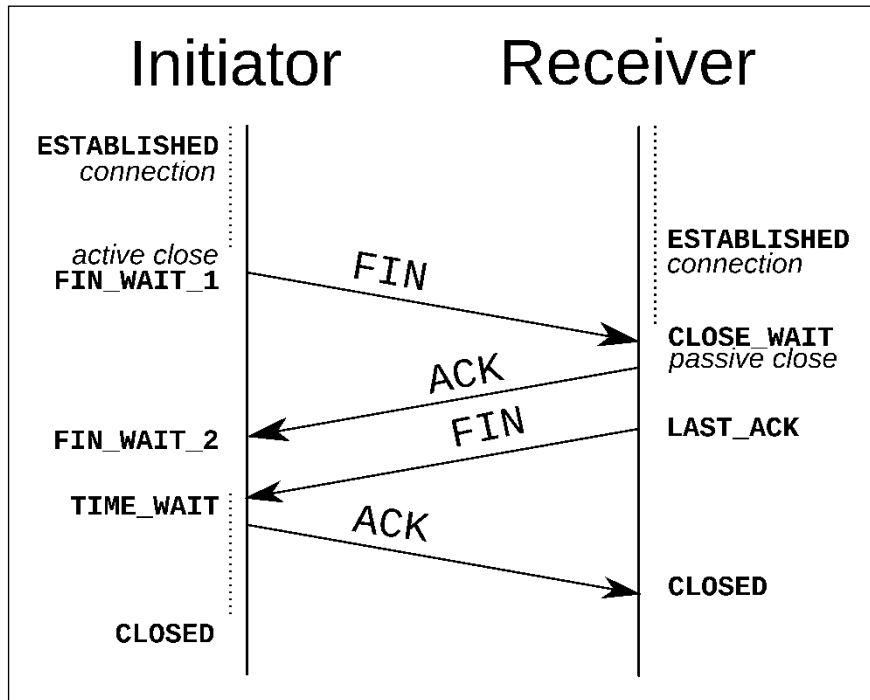
Integration testing

Integration testing involves more than one microservice component, so if two different components are integrated, a set of integration tests needs to be written to make sure they both integrate and exhibit the desired behavior.

Integration testing normally requires a simulation of a database schema or multiple components to be deployed in an environment and tested together. While a unit test can assert the behavior of the build binaries, an integration test is slightly more complex.

Mocking or stubbing can be carried out in order to simulate another application's endpoint behavior and assert if it is operating as expected.

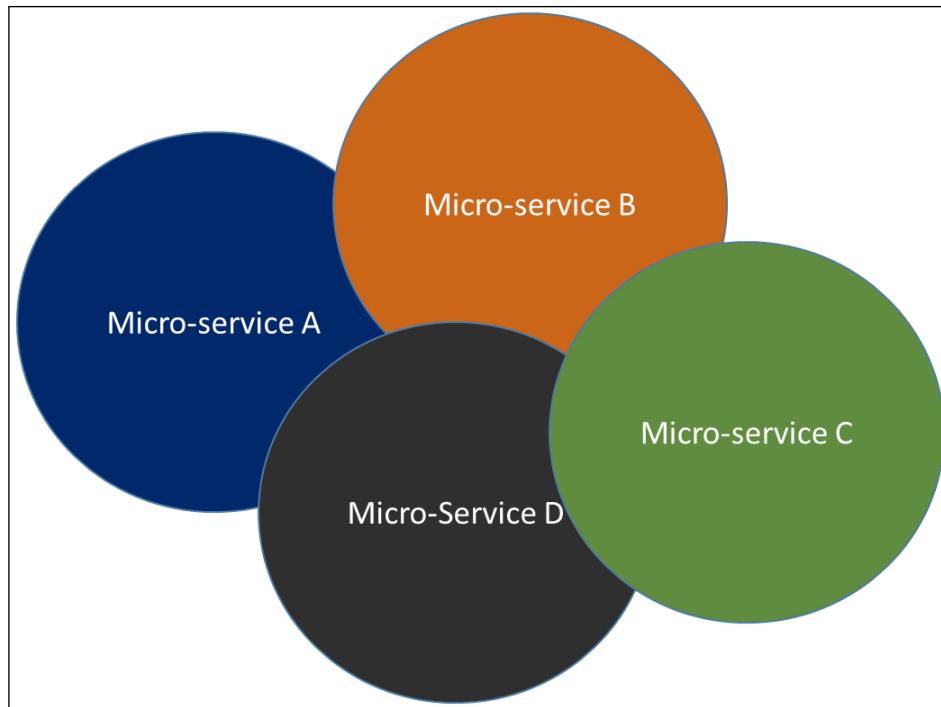
Integration testing could test that two different microservice endpoints can be connected and that a transaction such as a TCP handshake can be completed correctly between the initiator service and the receiver service, with the reception of ACK making sure that the two-way TCP handshake is working correctly between the two microservice applications:



System testing

System testing is normally carried out on a full-blown environment with a set of fully deployed components. System testing will test the whole system and is normally utilized as a final step before production. Some of the tests that can be carried out are user journey tests such as setting up a full transaction. This tests that the fully integrated system can pass all the end-to-end testing as a customer would use it in production.

This may result in integrating multiple microservice applications together such as microservice **A**, **B**, **C**, and **D** and making sure they all integrate functionally and work as a single entity:



Performance testing

Performance testing is fairly self-explanatory, it will baseline the application's performance on the first execution. It will then use that baseline to check for any performance degradations in the application every time a new release takes place.

Performance tests will be used to check performance metrics, this is useful to see if a code commit causes performance issues in the overall system. Performance testing can be incorporated into the system test phase.

Alternatively, performance testing can also mean **Stress Testing or Load Testing** the application, network, or infrastructure to its absolute limit and by writing tests to see to find if the system can cope with the desired traffic patterns.

Endurance Testing means setting a time period for testing and see how long the infrastructure, network, or application can cope with stress for a fixed period of time.

Spike Testing is making sure a system can cope with a sudden spike in traffic from a dormant traffic pattern, which tests if the system can cope with a high degree of variance.

Scalability Testing on the other hand can mean horizontally scaling out infrastructure or scaling up more applications to the point it makes no performance benefit. This identifies the scaling limits a system has.

Volume Testing can be used to see the volume of transactions or data a system can process over a given period of time.

The following diagram shows the different types of testing that fall under the performance testing umbrella:



User acceptance testing

User acceptance testing involves having end users test new features or functionality. User acceptance testing is normally utilized to make sure customers or product managers are happy with the development changes that have been made. This type of testing is normally exploratory and fairly manual. It is often used to test the look and feel of a website or graphical user interface.

Why is testing relevant to network teams?

Quality assurance is a huge part of network or infrastructure changes, it is not just solely a software development concern. If the network or infrastructure, which software is installed upon, is not operating as desired, then this will have the same customer impact as a software bug.

The customer doesn't differentiate between software bugs, infrastructure, or networking issues. All a customer knows is that they can't utilize products and as far as they are concerned the business is not meeting their needs or providing a good and reliable service.

Not having adequate testing can be very harmful to a business, as its very reputation can be damaged, and the rise of social media means that if websites are down or not operational, within a blink of the eye an outage can be all over social media channels.

If one user notices an issue they can send a tweet, which alerts other customers to the issue, one tweet becomes many and before the company knows it, the outage is trending on Twitter or other social media and now everyone across the world is aware that the business is having problems.

This situation is the worst fear for many online businesses, if the site is not up, operational, and providing a good user experience, then the business is no longer making money and customers may go to a competitor.

One of the key objectives for any development, infrastructure, or networking team is to provide a good service to end users and prevent downtime or outages. Typically, a set of **Key Performance Indicators (KPIs)** are used to quantify performance and set targets to decipher if a business is meeting customer needs.

So making the delivery of network changes less prone to error should be the aim of any network team. In *Chapters 4, Configuring Network Devices Using Ansible*, *Chapter 5, Orchestrating Load Balancers Using Ansible*, and *Chapter 6, Orchestrating SDN Controllers Using Ansible*, we looked at ways to automate network devices, load balancers, and SDN controllers using configuration management tooling. At the same time, having a set of repeatable tests for any network change should also be something that network teams are striving for. The ideal scenario being that a network team knows that a change is going to fail, before it has a customer impact. This means testing changes sufficiently before giving them a seal of approval and failing as fast as possible in test environments so breaking changes are not pushed directly to production environments.

One of the common concerns from network engineers when initially moving to an automated process is a lack of trust in the automation. Network engineers are used to going through due diligence and a subset of checks prior to releasing network changes. Just because automation is in place doesn't mean that the manual check-list network engineers used to validate network changes goes away.

However, when considering software delivery, the overall process is only as fast as the slowest component, so if those network validation checks remain manual, then the whole process will be slowed down. This will result in manual stops being placed in the automation process, which will inevitably slow down the delivery of a new product to market.

The simple solution is to automate each of the networking check-lists, so any validation that was carried out manually by a network engineer instead becomes an automated check or test as part of an automated test suite which is run alongside the automation.

These checks or tests are then written and built up over a period of time. So if a situation occurs when an edge case is found and it doesn't have test coverage that causes a failure. Rather than using the argument that the automation doesn't work and making a case to revert to tried and tested manual approaches, network engineers need to instead create a new test or check and add it to the automated validation pack which will catch the issue and fail in a test environment before it reaches the end user.

Network changes and testing today

Network teams still remain in the main work with a waterfall methodology, so they need to align and adopt a more agile approach. This will allow network teams to better integrate with the rest of the IT team and become a participant in the Continuous Delivery processes rather an observer.

When the waterfall methodology was the de-facto way of delivering software development projects to market, then a very rigid process lifecycle would be followed.

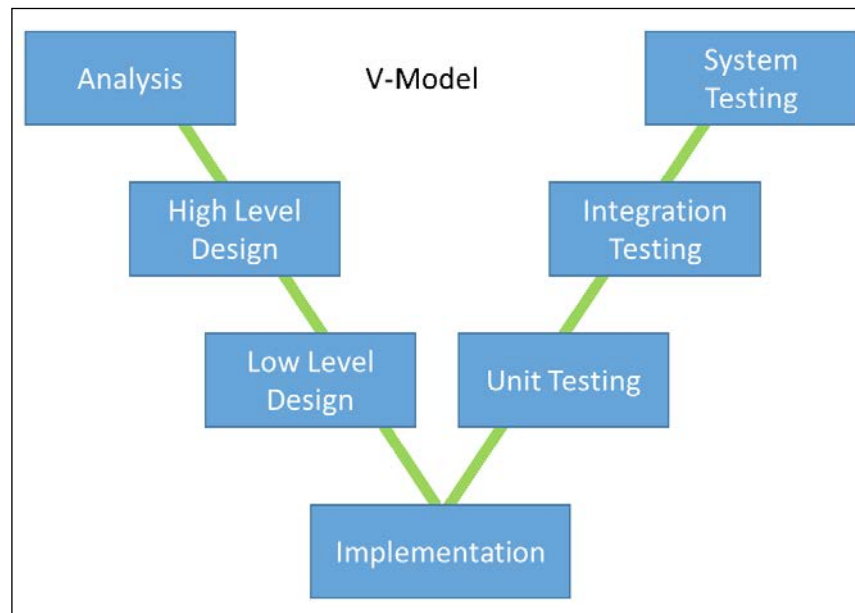
Waterfall processes stipulated every new feature would traverse the following phases:

- Analysis
- Design
- Implementation
- Test

One of the main implementations of the waterfall method was known as the **V-Model**, which was initially used to simplify projects into deliverable chunks. This meant that all stakeholders could identify progress and look for potential delays.

This simplification made project managers and senior management happy as they had an easy way of tracking projects and whether they were on time or going over budget.

The structure of the V-Model is shown as follows:



In waterfall terms, the analysis and design phases took place on the left-hand side of the V-Model and would happen at the start of the process. The left-hand side of the V-Model in simplistic terms is used to interact with stakeholders, do necessary research, and gather all necessary high-level and low-level requirements to work out what is required to implement a new product or change. The left side of the V-Model was also about documenting the overall process at an architectural level.

After the initial requirement gathering as part of the analysis phase, the analysis phase was signed off, which meant that the architectural design phase could begin and some meat could be put around the requirements. As part of the design phase, high-level and low-level design documents would be created to document the proposed changes which would have an associated review, sign off, and approval process.

Once the design was completed, then the left side of the V-Model was complete and implementation of the change or product would take place. The implementation phase could span numerous weeks, or even months, to deliver the desired result and this lifecycle phase sits at the bottom of the V-Model.

Once all the requirements were implemented, the implementation phase would be signed off and the project would move to the right-hand side of the V-Model where the Test phase would commence.

A test team would then carry out unit, integration, and then finally system testing on any change or new product feature. Any issues found with the implementation phase would result in a change request. This would mean that the high- or low-level design would need to be updated, re-work on the implementation would need to be done, and then tests would need to be repeated or re-written in order for the product to be refined.

With the move to agile development covered in *Chapter 3, Bringing DevOps to Network Operations*, the V-Model has been seen to be a sub-optimal delivery mechanism. For reporting purposes, the V-Model is ideal and transparent, but it means that the implementation process suffers from the rigid restrictions enforced on engineers.

The V-Model doesn't take into account that any engineer likes to iterate processes and the actual implementation they write down at the start of a process may not be the final design they implement. The V-Model doesn't align well to prototyping as engineers typically like to spend time with the system and try, fail, iterate, and then improve the implementation.

Not accounting for prototyping leads to multiple change requests which have cost implications to businesses, so using two week sprints in an agile methodology to plan in iterative development has proved much more realistic. Although it is still something that senior managers struggle with as they are indoctrinated with having the need to report due dates and milestones, the due date is, by all intents and purposes, a made-up date.

An engineer in the waterfall process will still do the same amount of prototyping to deliver implementations, and work takes x amount of time regardless of how a plan is structured. Agile development is just structured to accept prototyping and time-boxed spikes.

So the age-old question from a project manager to an engineer is always; *when will this be done by?* The engineer that replies; *I don't know* isn't an acceptable answer in a waterfall methodology. What is expected is an estimate, or in engineering circles a made-up date, which the project manager will likely change later when, inevitably, said date isn't met.

So how does any of this have any relevance to network changes and testing overall? Well, network teams today typically implement a mini V-Model when they think about making network changes. Network managers will act as a project manager that will plan out a design, implementation, and test phase cycle and report this back to senior management teams as network changes are seen as long pieces of work that need massive planning and testing before implementation.

Network managers may not split out testing into test, integration, and system testing as traditionally, network testing is not as sophisticated as this, but it doesn't allow network engineers the freedom to prototype.

Instead network engineers, like infrastructure engineers or any operational team before them, will be pressured into making changes to a rigid plan. The plan will be indicative of the following criteria being met:

- Does the implemented change work as desired?
- Did the change break anything?
- Is the documentation updated to reflect change?

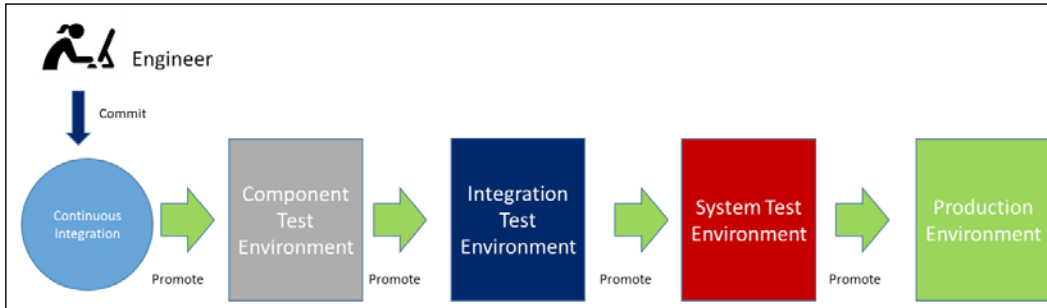
If all these points are met, they would have deemed a successful change by a network team.

However, this doesn't tell the complete and whole story as other points have to be considered when making network changes within the remit of a Continuous Delivery model:

- Will the change break anything that isn't immediately visible?
- Was the same change implemented to pre-production environments?
- Was the same change tested and validated on pre-production environments?

The initial three points are mandatory requirements when doing any change, but the remaining three points should be considered mandatory too, in order to maintain a successful Continuous Delivery model. This is something of a mind-set change for network engineers, that they need to take this into consideration when making changes.

All network changes from a network engineer need to be committed to the source control management system and propagated through all the necessary environments before being pushed to production, as shown in the following figure:



If a change is made directly to production manually by network engineers and not implemented first on test, pre-production and production environments via an automated, then the network configuration will be forever misaligned on test environments. This will have dire consequences as the configuration of test, pre-production environments, and production will have drifted apart.

This means that any developer, infrastructure, or network engineer using test and pre-production environments expecting a copy of production for mission-critical changes will be sadly disappointed. This can, in turn, compromise any tests run on those environments as they are no longer a proper reflection of the production estate.

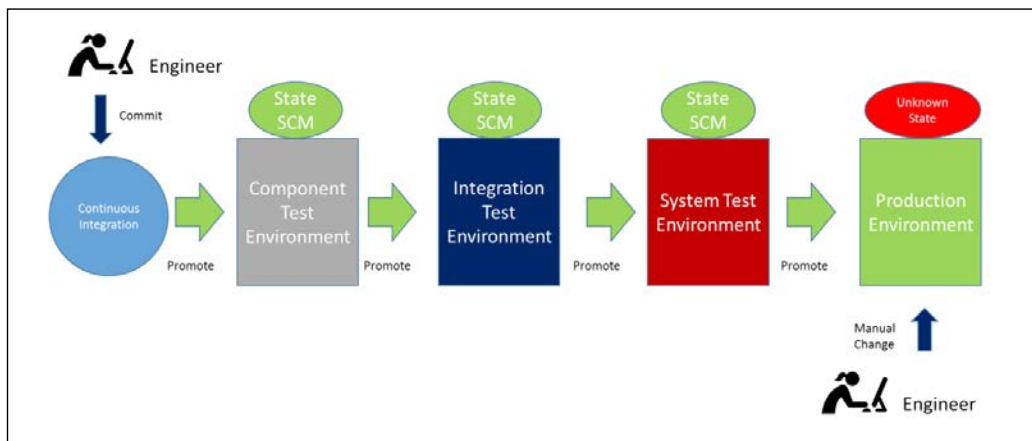
So what does this mean in practice? The system test box on the V-Model may pass in pre-production environments but fail in production. This does not build confidence in the Continuous Delivery process which is now a mission critical part of the business.

It cannot be highlighted how important it is to make sure all network, code, or infrastructure changes are pushed to pre-production environments prior to production to maintain the validity of all environments. Any team deviating from this process can compromise the whole system and negate the testing.

This is not only used to test the changes, but also to keep the pre-production environments as a scaled-down mirror image of production that avoids the scenario of all tests pass in test environments but when they are deployed to production they cause outages to customers. So it means it is important that all validation tests are completed in associated test environments before a change is released to production.

If manual changes are pushed directly into production, even in the event of emergencies, then the changes need to be immediately put back into the **Source Control Management (SCM)** system, the SCM system should be the single source of truth for all configuration at all times.

If any manual changes are applied, snowflake environments will become common, which are shown below. This is where an engineer has made a manual change to production outside the process and not pushed the change to any of the other environments using the deployment pipeline:



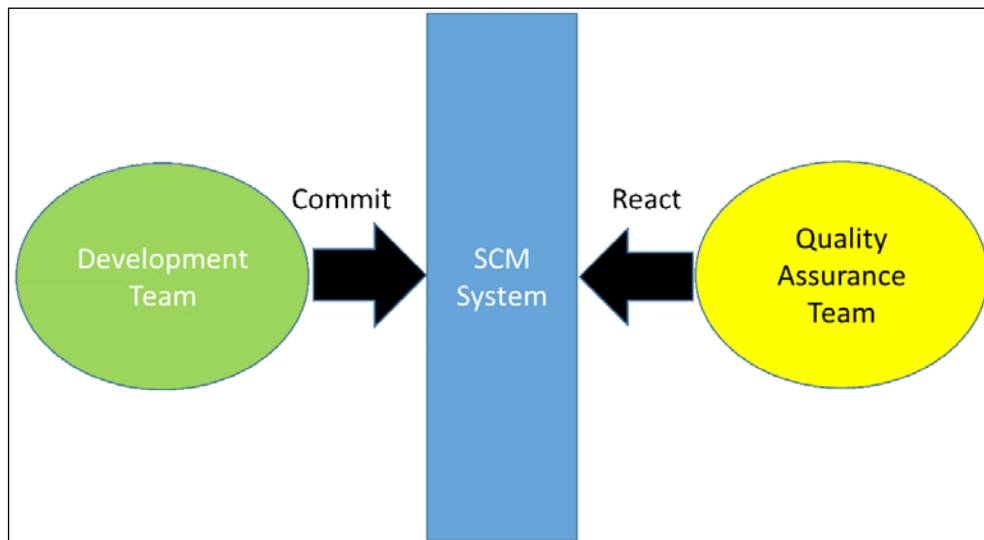
In order for network changes to be delivered at the desired rate, network changes and testing cannot continue to be done in a mini V-Model strategy. If network teams and managers are serious about being collaborators in Continuous Delivery models and DevOps models, they need to keep pace with the rest of the agile changes being made in development and infrastructure teams.

The solution though is not to stop validating changes and paying due diligence, lessons can be learned from formulaic quality assurance processes that have been successfully applied on development and infrastructure changes for years, these processes can also help test network changes.

Quality assurance best practices

Quality assurance teams, when utilizing a waterfall V-Model structure of delivery, worked in silos that retrospectively tested development changes once they were completed by a development team.

This led to quality assurance teams having to react to every development change, as it was an impossible task having to write tests for a feature they had not yet seen, or understand how it fully operated. Situations would often arise where developers without warning would commit features into source control management systems and then quality assurance teams would have to react to them:



This method of working provided lots of challenges such as:

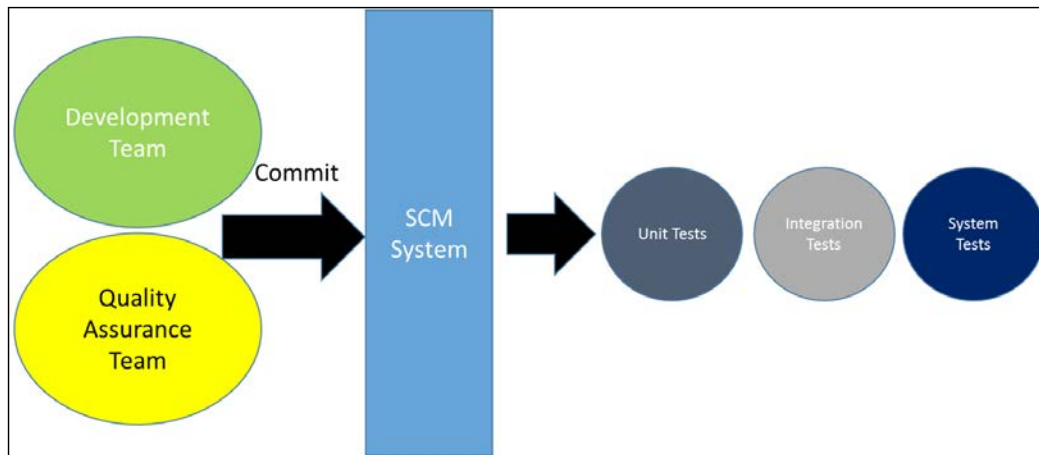
- Developers changing user interfaces, so the quality assurance team's automated tests broke as test engineers were not aware of the user interface changes
- Test engineers not understanding new features meaning appropriate tests weren't written to test functionality properly
- Developers having to spend lots of time explaining how features worked to quality assurance testers so they could write tests post-commit
- Delays associated with fixing broken regression tests that were not down to bugs but test issues
- Quality assurance teams were acting in a completely reactive fashion as they could not see what new developer changes were coming
- Quality assurance packs never passing, or being green, meant that actual software issues slipped through the process

So, when considering network testing, the solution to this problem is not to hire a separate test team. Instead, it is about incorporating and integrating network testing into a Continuous Delivery model.

Agile development has shown that as code changes were being written, embedding quality assurance test engineers in the development team meant that tests could be written pre-commit. It is a far more productive method of working.

Moving quality assurance engineers out of the siloed quality assurance team and allowing them to work together in the same scrum team means that individuals that work together can collaborate and make sure that the submitted commit will work at every phase.

The associated regression, integration, or system tests then form a set of automated quality gates that the change will propagate through:



The main benefits of the agile testing approach over using a siloed waterfall approach is:

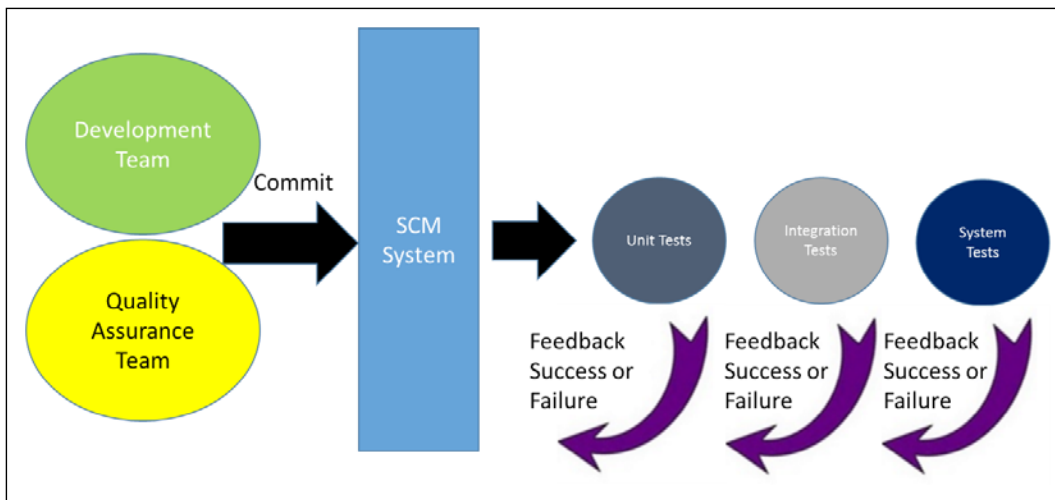
- Quality assurance testers were no longer working reactively and have complete visibility of what developers are creating
- Each agile user story could have proper acceptance criteria written that included automated testing, allowing quality assurance engineers to work on test tasks as developers code new features
- When new features were coded, relevant tests are written for a new feature
- New feature tests can be added to the regression pack, so that every time a code commit is made the feature is tested

This process change removes the inhibitor, which is simply the team structure, and joins two teams together so they become more productive, which is in essence the DevOps way.

Creating testing feedback loops

If we think back to the continuous integration process in *Chapter 7, Using Continuous Integration Builds for Network Configuration*, then we had the commit process. The commit essentially starting the whole Continuous Delivery process. Once a commit has taken place, the change is already on the road to production. Any commit to the trunk/mainline/master branch is a final change, so if a network commit is made, it is already on its way to production.

If no validation engine or tests exist post check-in, then changes will flow all the way through test environments reaching production environments.

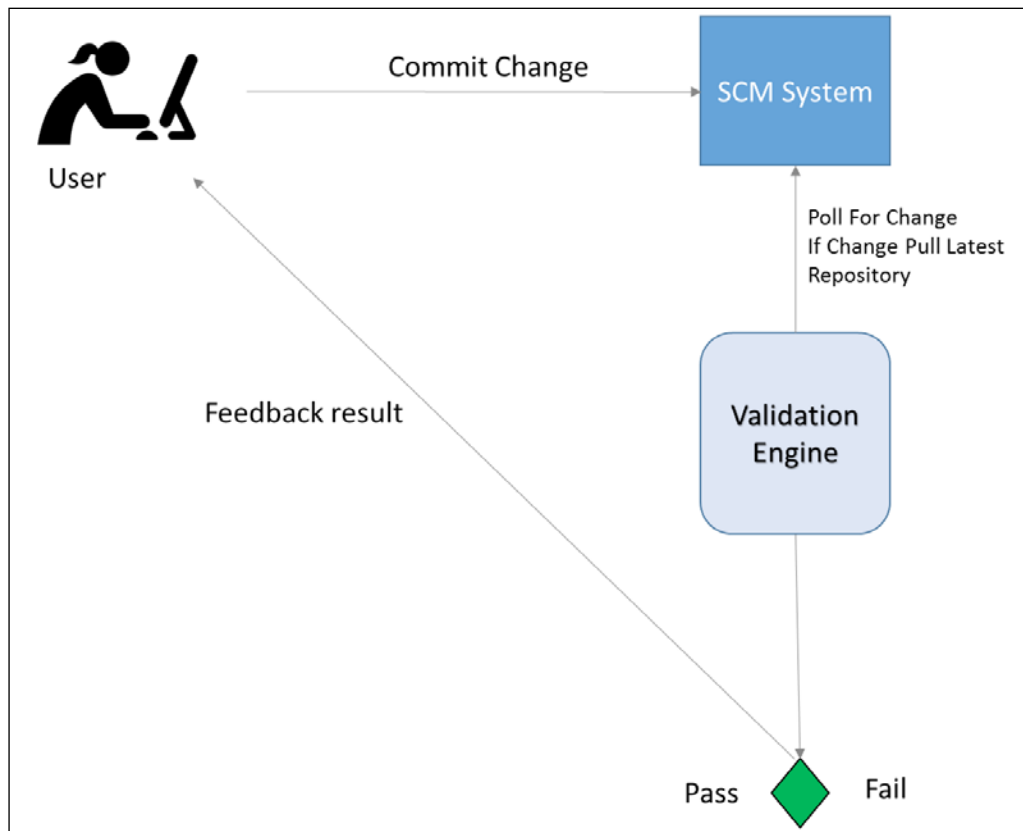


This means that utilizing feedback loops with proper test gates is essential, so once a code commit has taken place, it will be adequately tested and provide an immediate indicator that a change has failed. Once all the quality gates have completed successfully only then should the change be promoted to production, this model promotes continuous improvement and failing fast. The further to the left a change fails, the less cost it incurs to a business.

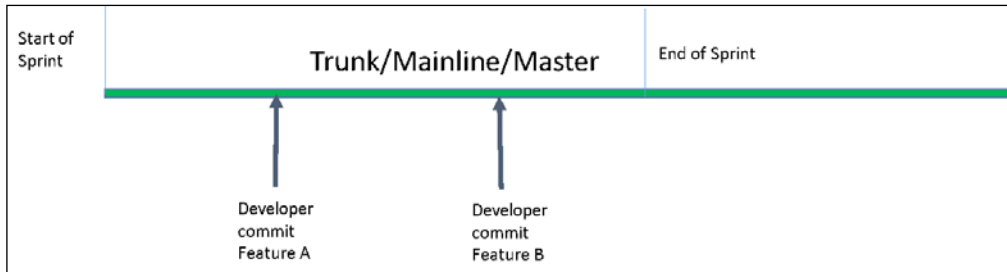
Continuous integration testing

In *Chapter 7, Using Continuous Integration Builds for Network Configuration*, we focused on the process of continuous integration and how multiple different checks can be applied as part of the validation engine for a user commit. This makes sure that the user commit is always properly validated.

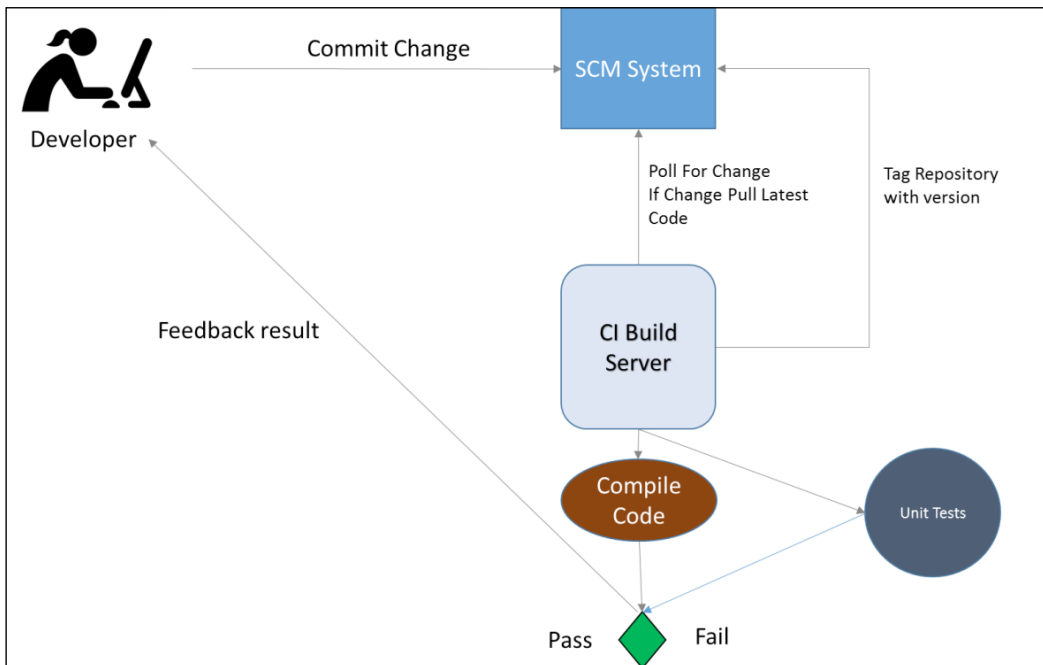
Continuous integration provides a set of feedback loops where a code commit is submitted to the SCM and the validation engine will return either a pass or a failure. All testing can form the validation engine for changes:



The continuous integration process when applied to development, takes the approach that all changes are committed to the **Trunk/Mainline/Master** branch:



The new development feature will be committed to the **Trunk/Mainline/Master** branch. This new commit will be compiled and be immediately integrated with the rest of the code base, and then subsequent unit tests will then be executed to determine a pass or failure against the build binaries, as shown in the following diagram:



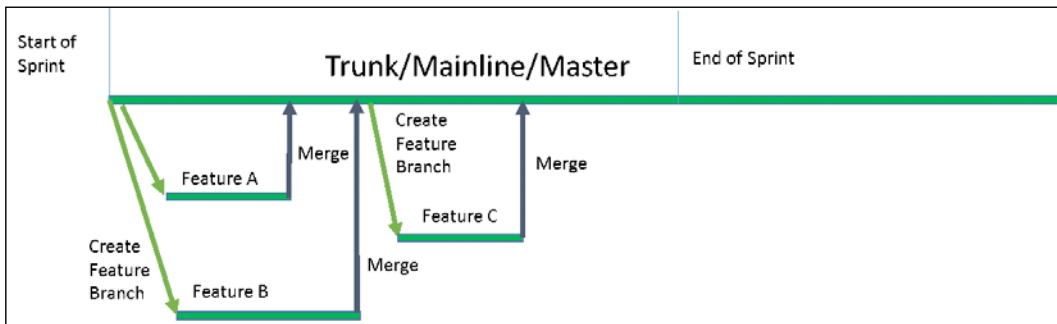
Using the commit to **Trunk/Mainline/Master** continuous integration approach relies on a degree of discipline from teams. If a commit fails and the **CI Build Server** returns a failed build, then the team member that made the failed commit has a duty to fix the build immediately, by either reverting or fixing the broken commit.

Continuous integration builds should under no circumstances ever be left in a failed state as it means the **Trunk/Mainline/Master** is not in a clean state and all subsequent code commits will not have valid continuous integration performed until the build is fixed. This slows down a team's productivity so continuous integration is a collaborative process and failure should be seen as a learning opportunity.

Gated builds on branches

Another popular method is using **feature branches** and **gated builds**. Every time a developer makes a change, they will raise a merge request which will be peer-reviewed by other members of the team and then subsequently merged.

Each merge request, when accepted, will start the merge process, but as part of the merge process something known as a gated build will execute.



A gated build will be invoked when a merge occurs prior to integration with the **Trunk/Mainline/Master**. It will run the equivalent of a continuous integration build as a pre-commit, but only if the build and unit testing associated with the pre-commit build passes, will the contents of the merge request be merged to the **Trunk/Mainline/Master** branch.

The gated build process means that the **Trunk/Mainline/Master** branch is always kept completely clean and functional. Where pure continuous integration can have developers break the continuous integration build, gated builds prevent this from happening, as long as the tests are good.

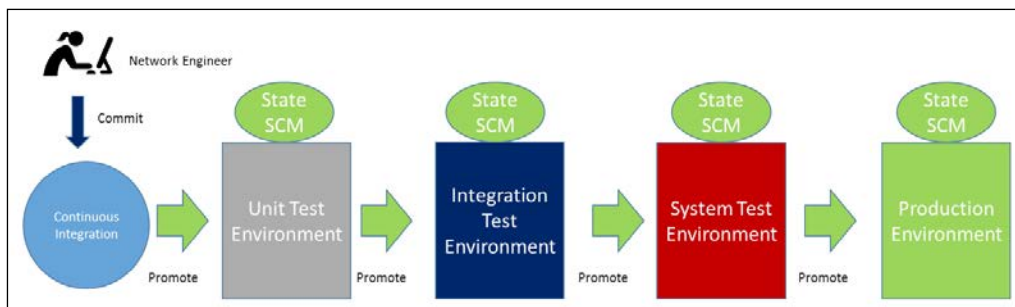
Applying quality assurance best practices to networking

Network teams can greatly benefit from adopting some of the best practices and tried and tested methodologies that have been implemented to test development or infrastructure changes.

Quality assurance is all about principles and processes, so test methodologies are fairly agnostic and the tools used to implement the process come secondary.

When teams are working within a Continuous Delivery model, any changes to network devices, load balancers, or even SDN controllers should be defined in source control using orchestration and configuration management tools such as Ansible.

In a Continuous Delivery model, network changes need to propagate through environments and at all times be governed by source control management systems, with the state of the SCM system being the state of the network:

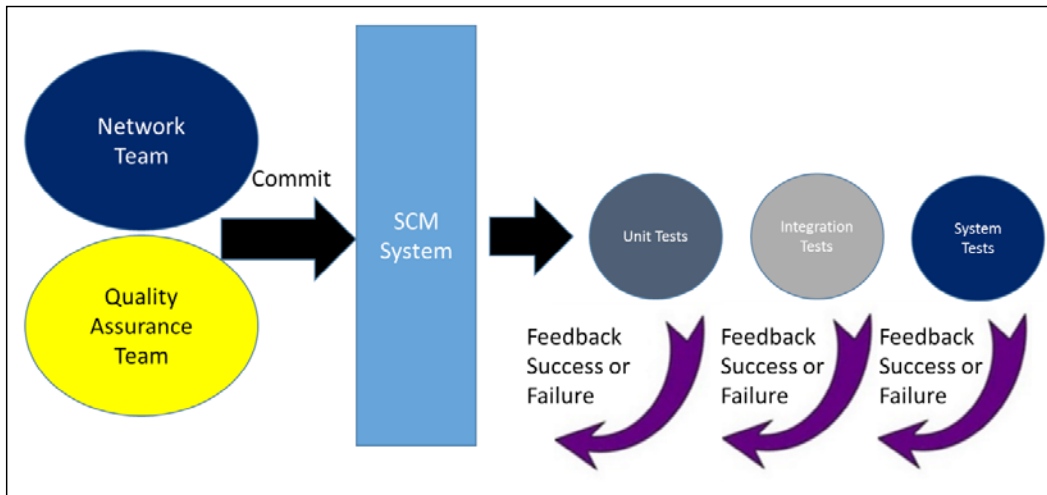


Network changes can be treated much like code changes and adequate testing could be created as a network team initiative or by collaborating with the quality assurance team. The type of testing at each phase may vary slightly from the set of tests that a development or infrastructure team would run as part of their deployment pipeline.

However, equivalent network-specific testing can be derived to create a set of robust tests that network changes have to traverse before being deployed to production by associating particular network tests with each quality gate on the deployment pipeline.

Network teams, such as development and infrastructure teams need to create a set of feedback loops to govern network changes, so that different test categories can be executed in the deployment pipeline.

All testing should ideally be automated as part of each network change, in a proactive manner and written at the same time as the network change is being lined up. This then allows network changes to be tested in an automated manner at the point of inception:



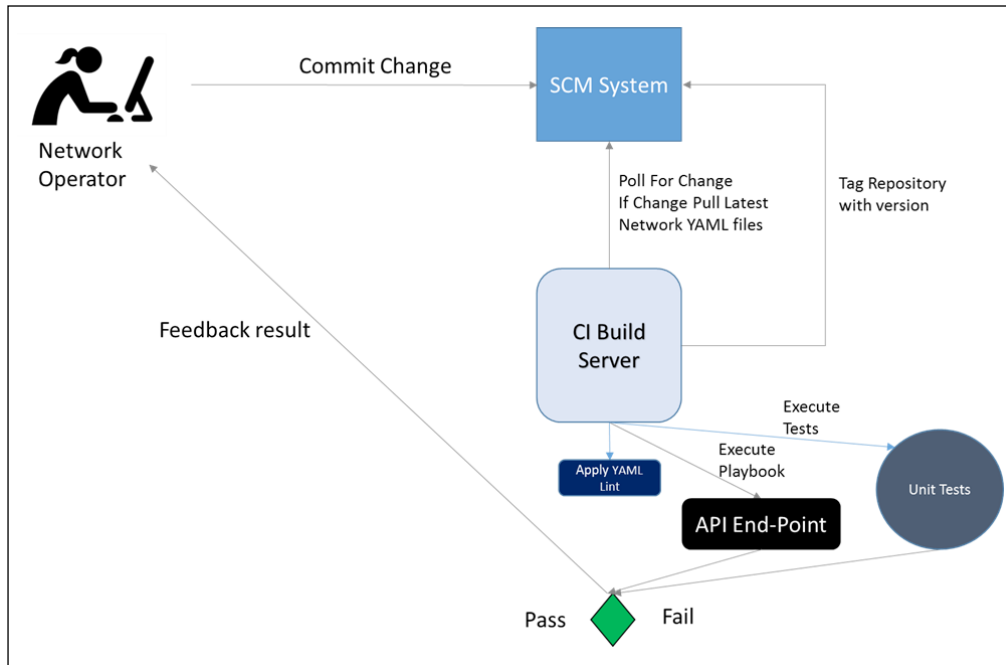
When setting up continuous integration, selecting either continuous integration or a gated build strategy is down to the preference of the network team or engineers that commit changes.

Unit testing should be integrated with the network continuous integration process. A network operator will first check in a code change or change the state of the network.

The **CI Build Server** will check the Ansible `var` YAML files using Lint, which will make sure that the YAML files are valid syntax.

If valid, the same playbook that would be executed on any downstream environment will be executed against a CI test environment to make sure the playbook is successful in terms of syntax and execution.

Finally, a set of unit tests will be executed against the environment to validate its functional and desired state of the environment after the playbook has been executed:



The important thing to note is that unit tests are executed as part of the continuous integration process and that these tests can either be part of the merge request validation or executed on commit to the **Trunk/Mainline/Master** branch.

Assigning network testing to quality gates

When looking at what type of testing network teams can carry out to validate network changes, they can be broken into different test categories and assigned to different quality gates.

Some of the main test environments covered in this chapter are:

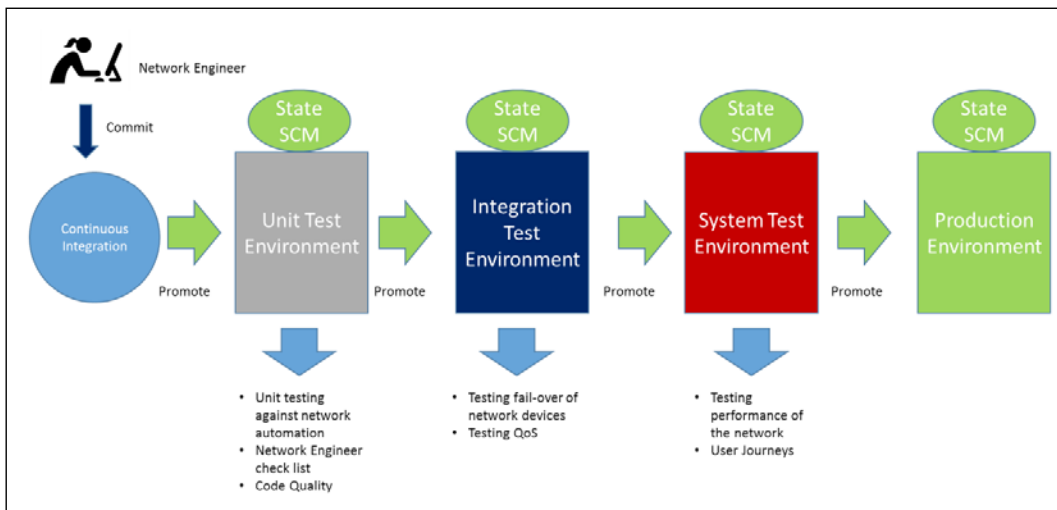
- Unit test
- Integration test
- System test

Before considering where to put tests, we should first look at the network team's needs. With a blank canvas what would be a beneficial set of tests that could help with network operations?

Some of the following tests spring to mind, but any check or validation that is valid to a particular team is applicable and should be included:

- Network checklist that network engineers carry out manually when making changes
- Unit testing against network automation to make sure the network device is in the desired state
- Testing performance of the network to see what the desired throughput is and test when parts of the network become oversubscribed and need to be scaled out
- Testing failover of network devices
- Testing network code quality
- Testing different user journeys through the network
- Testing Quality of Services

All of these types of tests can then be assigned to particular test environments and quality gates created:



Available test tools

Test tools, like all tools, should be used to facilitate test processes and outcomes. So, for every single test quality gate, tools are required to wrap processes, schedule, and execute tests.

There are various test tools available on the market today that network engineers could greatly benefit from using.

Unit testing tools

Network unit testing as said many times before will form part of the continuous integration build process and scheduled by a continuous integration build server.

One open source tool that can help with unit testing network changes is Test Kitchen. **Test Kitchen** is a unit testing tool which utilizes the **Busser** framework and can be used to carry out infrastructure testing. Test Kitchen supports many test frameworks such as **Bats** and **RSpec**.

The **Test Kitchens Busser** framework is comprised of the following architectural components:

- Driver
- Provisioner
- Platform
- Suites

Test Kitchen defines all its plugins using a `kitchen.yml` file, which outlines the Driver, Provisioner, Platform, and Suites to use for the testing.

A **Driver** can be any platform that can be used to provision a virtual machine or container. Test Kitchen has support for Vagrant, Amazon, OpenStack, and Docker, and so can be used to test infrastructure changes.

A **Provisioner** is a configuration management tool such as Ansible, Chef, Puppet, or Salt and is used to configure the server into the state that needs to be tested.

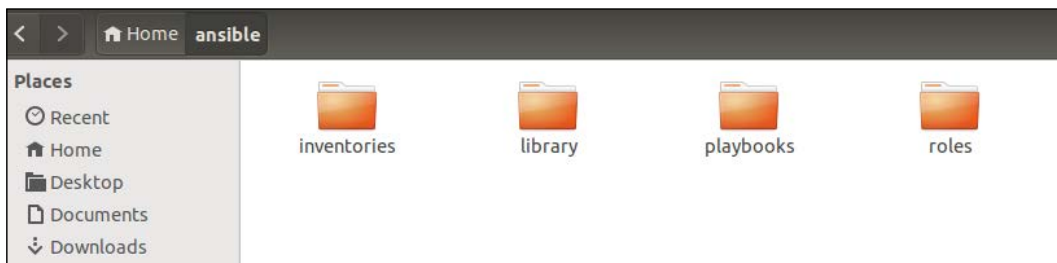
The **Platform** is the operating system that the Provisioner will execute on. Multiple Platforms can be specified for cross-operating system testing. This could be very useful when testing new versions of network operating systems operate in the same way as their predecessors when doing software upgrades.

Suites are used to create a test suite in combination with the Platform definition, so if two different Platforms are defined, then unit tests will be executed against each different platform in a consistent manner.

Test Kitchen example using OpenStack

The `test_kitchen` gem will need to be pre-installed on the Ansible controller host. Then perform the following steps:

1. From an Ansible controller node, in the folder containing the top player file structure, as shown in the following screenshot:



Here, execute the following command:

```
kitchen init -provisioner=ansible -driver=openstack
```

This creates a `kitchen.yml` file and a `test` subdirectory.

2. Next, the folder `test` folder needs to be created which will store the unit tests:

```
mkdir ./tests/integration/default/bats
```

3. The `test_kitchen` file will then need to be populated with the Driver, Platform, Provisioner, and Suites.

In the following example:

- The Driver is specified as OpenStack, with a `cumulus-vx` image and Platform being created.
- The size of the image is `m1.large` which specifies the CPU, RAM, and disk for the server.
- The instance will be created within the `network_team` tenant and `qa` availability zone.

- Once spun up, the `configure_device.yml` playbook will be executed to configure the network device before the default folder under `test/integration` which was defined in step 2. This tells Test Kitchen the location of the Bats tests that will be executed to test the state of the device:

```
---
driver:
  name: openstack
  openstack_username: admin
  openstack_api_key: *****
  openstack_auth_url: http://10.102.100.129:35357/v2.0/tokens
  image_ref: cumulus-vx-2.5.3
  flavor_ref: m1.large
  openstack_tenant: network_team
  availability_zone: qa
  server_name: network_unit_testing
  network_ref:
    - net-unit-testing
  key_name: provisioner

provisioner:
  name: ansible_playbook
  playbook: ./playbooks/configure_device.yml
  hosts: localhost
  require_ansible_repo: true
  modules_path: /library
  extra_vars:
    environment: ci

platforms:
  - name: cumulus-vx-2.5.3

suites:
  - name: default
```

4. Each test can be given a unique name and the `.bats` file extension to define each unit test under the `bats` directory that was created in step 2:
`Test/integration/default/bats/unit_test.bats`

5. An example of a test that can be written using Bats is as follows:

```
@test "network eth0 interface is up" {  
  run sudo ifup eth0  
  [ "$status" -eq 0 ]  
}
```

This checks that the `eth0` interface is in a good working state when executed.

6. Finally, to execute `test kitchen`, execute the command shown in the following screenshot:

```
kitchen test
```

Test Kitchen will then carry out the following workflow:

1. Create instance in OpenStack.
2. Run playbook.
3. Install Busser plugin.
4. Run unit tests.
5. Destroy instance if all tests passed.

Network checklist

Network engineers, as discussed, often have a set of manual checklists that they use to validate if a network change has been successful or not.

Sometimes, this could involve validating whether a user interface has the desired configuration that checks if the automation has worked as desired.

Instead of doing these checks manually, Selenium can be used to carry out graphical user interface checks.

Selenium's workflow can be summarized as test scripts invoking the Selenium web driver which then creates a browser session to test a website or web page:



Test scripts can be written in multiple languages such as Java, Python, or Ruby.

Selenium can be installed in Python form by doing a pip install when using Python for authoring scripts.

As Selenium is browser-based, it works with multiple browsers such as Internet Explorer, Firefox, Chrome, and Safari and tests cross-browser support.

A Selenium test sample is shown in the following screenshot; this script will launch google.co.uk in Chrome, type DevOps For Networking and finally click the **Search** button on Google:

```
from selenium import webdriver
from selenium.webdriver.common.by import By

driver = webdriver.Chrome('./selenium/webdriver/chrome/chromedriver')
driver.get('http://www.google.co.uk')

q = driver.find_element(By.NAME, 'q')
q.send_keys('DevOps For Networking')
q.submit()
```

So, any graphical interface can be screen scraped such as a load balancer or network device interface to assert that the correct information has been entered and returned. This can also be useful if older network devices don't have an API.

Network user journey

A good test methodology is to test user journeys throughout the network. This can be done by doing point-to-point testing in the network.

A good example of network user journeys may be testing **Equal Cost Multipath (ECMP)** on Leaf-Spine architecture to make sure it is performing as desired.

Another test may be setting up point-to-point tests across data centers to make sure links are performing as desired and do not suddenly depreciate.

Setting up user journey testing means that if a baseline performance drops, then it can be tracked back to specific network changes as part of the network deployment pipeline. This is done in much the same way as baselining application performance and making sure a new release doesn't cause a drop in performance that will impact end users.

Network user journey testing mean that if an ill-performing path through the network is found, then it can be localized and fixed quickly so it improves mean time to resolution when issues occur. Network engineers can use a tool such as **iPerf** to send large amounts of packets through points in the network. This can be useful to see where the bottlenecks are in the network and make sure the performance is as desired.

Quality of Service

A lot of network tools now offer **Quality of Service (QoS)**, which allows network operators to limit the amount of network bandwidth that particular tenants utilize in a network.

This prevents noisy test environments from impacting a production environment. This is possible as network devices can set guarantees on performance on particular tenant networks. This means that certain application workloads are always guaranteed a certain network throughput, while other less crucial tenant networks can be capped at peak times.

Different thresholds and alerting can be set up on network devices and faults in network hardware can be detected if the QoS drops at a random time. It also guards network engineers against the age-old: *I think we have a network problem*. Instead they can prove it is an application issue, as the network service is stable and performing as desired and can be easily displayed.

It is good to simulate and test QoS away from production environments and have network teams come up with different scenarios to design the best fit for the network, based on the applications that they are hosting.

Failover testing

Failover testing should ideally be tested regularly by network teams, as modern networks should be disaster recovery-aware and designed for failure.

Network failover tests can be simulated by writing an Ansible playbook or role that disables a service or reboots a switch to make sure that the system adequately fails over.

Utilizing `delegate_to: localhost`, API commands can be issued to network devices such as switches to disable them programmatically using the API. Alternatively, Ansible can SSH onto a network device's operating system and issue an impromptu hard reboot.

Supplementary monitoring should be set up while doing failover testing to make sure the network does not drop packets and test the speed at which the network device fails over after the initial primary device is disabled.

Network code quality tooling

When defining the desired state of the network as code, make sure the Python code that is written to create Ansible modules, as well as any other code that is used is of a high standard and good quality.

SonarQube is an open source code quality tool which allows teams to analyze their codes quality. Its architecture is comprised of three main components:

- SonarQube Runner
- SonarQube database
- SonarQube web interface

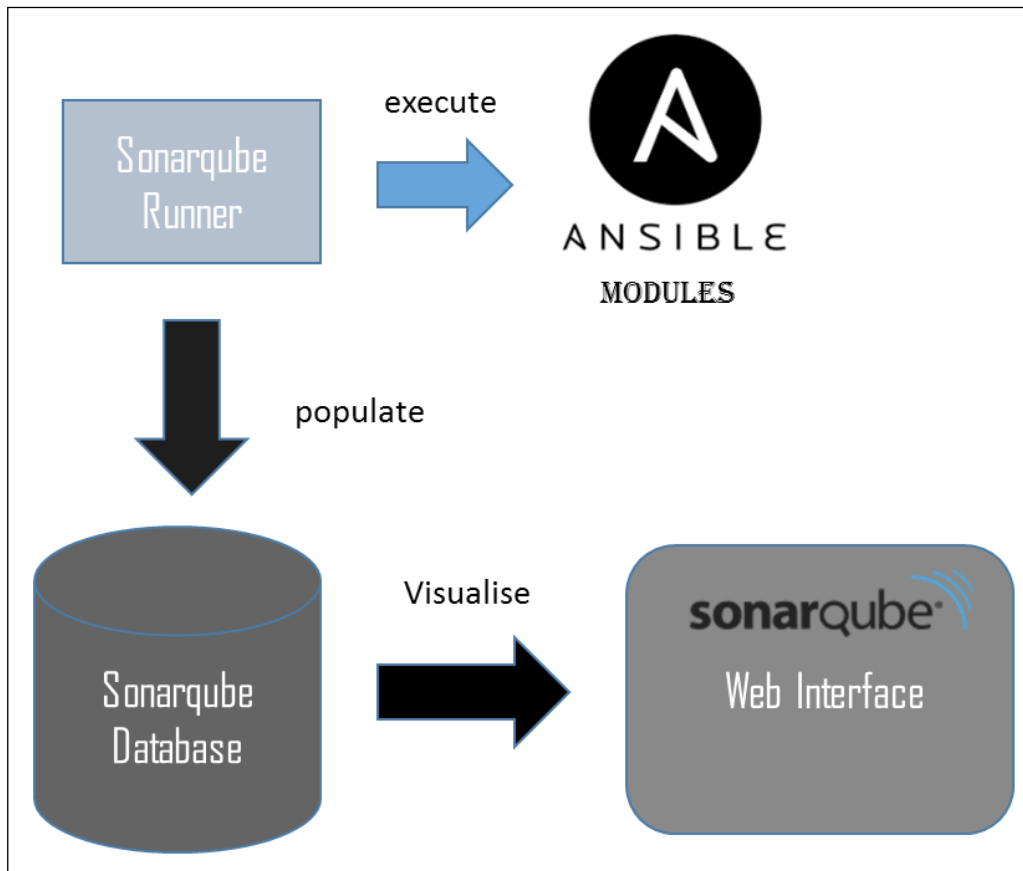
Sonar has a range of plug-ins that can be configured to provide unit test reporting, code coverage, or code quality rules and can be set-up for any language be it Python, Java, or C#.

SonarQube will snapshot a code repository every time it is run and store the history of a project in terms of code quality. This can be trended over time showing quality improvements or drops in the code quality. Sonar can be used to define specific best practice or rules, which show up as violations when broken by commits.

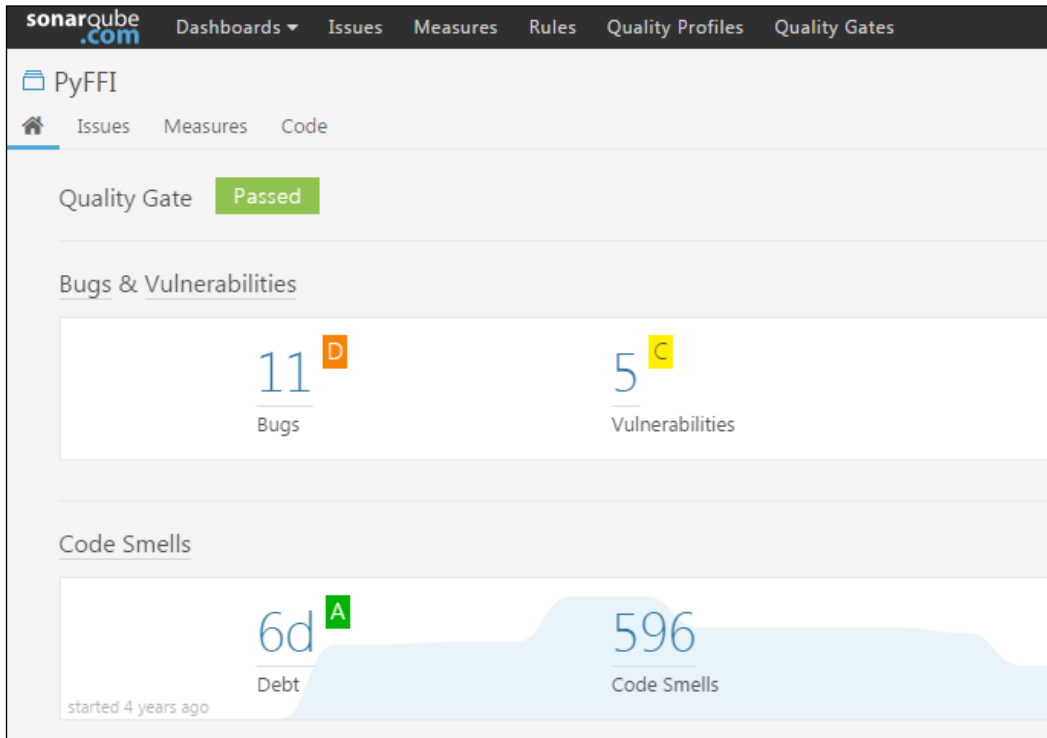
The **SonarQube Runner** uses a `sonar.properties` file at runtime that can be included as part of the source control management system. This can be pulled down as part of the continuous integration process. This means that after a new code commit on a custom Ansible module the SonarQube Runner can be executed against the code to test the new commit and see the impact.

The SonarQube Runner will execute a code quality check using one of the plug-ins stipulated in the `sonar.properties` file. In the case of a new or changed Ansible module that will invoke the Python-specific group of code quality tests. Information will subsequently be displayed on the Sonar web-interface once the analysis is complete.

The workflow for this process is shown in the following screenshot with the SonarQube Runner triggering the whole process:



An example of the sonar Python SonarQube project dashboard is shown in the following screenshot, outlining the bugs, vulnerabilities, and tech debt to fix all the issues in the code:



Tracking code quality and metrics is very important when implementing a continuous improvement model in any company. So adequately measuring and analyzing where improvements can be made in the code that drives all processes is important in order to have engineers engage and write tests.

Summary

In this chapter we have looked at why testing network changes are necessary. We focused on the benefits of utilizing feedback loops to continuously improve network operations. We then explored some of the challenges associated with the way network teams approach network changes and testing and how they will need to adapt and adopt quality assurance best practices to keep up when companies are running a Continuous Delivery model supplemented by a DevOps methodology.

We then looked at how network teams could set up quality gates for testing and looked at some of the tests that could be mapped at each stage of testing. Finally we looked at some available tools that could be used to carry out network testing to implement unit testing, check-lists, and code quality checks.

In this chapter you have learned about different types of test strategies such as unit, component, integration, performance, system, and user acceptance testing. Key takeaways also include quality assurance best practices, and why they are applicable to networking and different types of network validations that could help assert automated network changes.

This chapter has also delved into test tools that can be used to help test networking such as Test Kitchen (<http://kitchen.ci/>), SonarQube (<http://www.sonarqube.org/>), and iPerf (<https://iperf.fr/>).

In the next chapter we will focus on deployment pipelines, look at the tooling that can be used to automatically deploy network changes. We will also look at the difference between Continuous Delivery and deployment and when each approach should be implemented.

The following blogs and presentations may be useful for further understanding microservice test strategies in more detail:

- <http://martinfowler.com/articles/microservice-testing/>
- <https://www.youtube.com/watch?v=FotoHYyY8Bo>

9

Using Continuous Delivery Pipelines to Deploy Network Changes

This chapter will focus on some of the different methods that can be used to deploy network changes using deployment pipelines.

It will first look at Continuous Delivery and continuous deployment processes and what these methodologies entail in terms of workflow.

We will also look at the different deployment tools, artifacts repositories, and packaging methods that can be used to set up deployment pipelines and ways in which network changes can be integrated into those pipelines.

In this chapter, the following topics will be covered:

- Continuous integration package management
- Continuous Delivery and deployment overview
- Deployment methodologies
- Packaging deployment artifacts
- Deployment pipeline tooling
- Deploying network changes with deployment pipelines

Continuous integration package management

In *Chapter 7, Using Continuous Integration Builds For Network Configuration*, we looked at the process of continuous integration and in *Chapter 8, Testing Network Changes*, we looked at adding testing to the continuous integration process to provide increased validation and feedback loops in case of failure.

When carrying out continuous integration, using a fail fast / fix fast philosophy is desirable. This involves putting in necessary validation checks to decipher whether a build is valid and provide feedback loops to users.

This promotes the correct behavior within the teams that do frequent, small, incremental changes, which de-risks the changes. While each change is validated using the **Continuous Integration (CI)** engine with instant feedback on changes, a process of continuous improvement is adhered to as teams strive to make more robust solutions that will pass all quality checks.

As important as providing feedback loops is, producing successful builds is equally important to the process as this is how products are shipped to market. When a continuous integration build completes, it often needs to package build artifacts that are in a fit state so they can be deployed to target servers. This is often referred to as creating a shippable product or artifact.

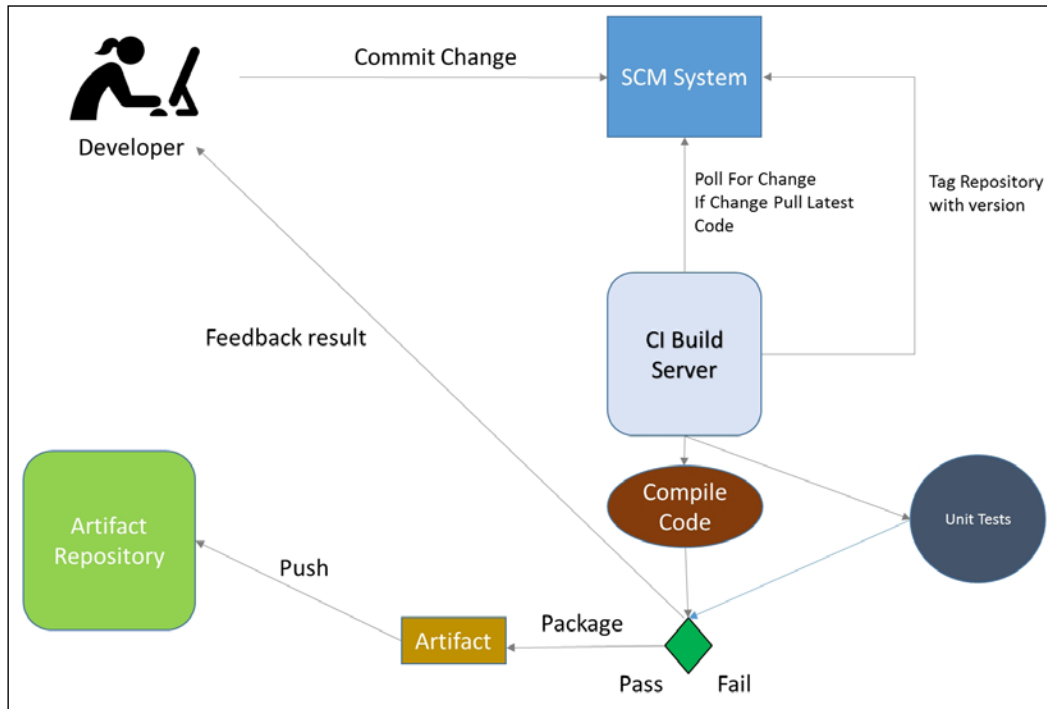
Any continuous integration process should carry out the following steps:

- Commit
- Build (Compile/Version/Tag)
- Validate
- Package
- Push

Every time a new commit takes place, a new continuous integration build will be triggered. This will result in a code being pulled down from the SCM system, which will trigger a build step, which can either be a compilation process, or if the build process is not using a compiled language, then versioning or tagging of the binaries. Finally, a set of validation steps will be carried out inclusive of any required testing.

If all validations prove successful, then a set of post-continuous integration process steps need to be carried out. Post-build steps will include the package and push process, this means packaging build binaries and pushing the newly versioned package to an Artifact Repository of choice.

An example process that includes **Commit Change**, **Build (Compile Code)**, **Validate (Unit Tests)**, **Package (Artifact)**, and **Push to an Artifact Repository** is shown in the following diagram:



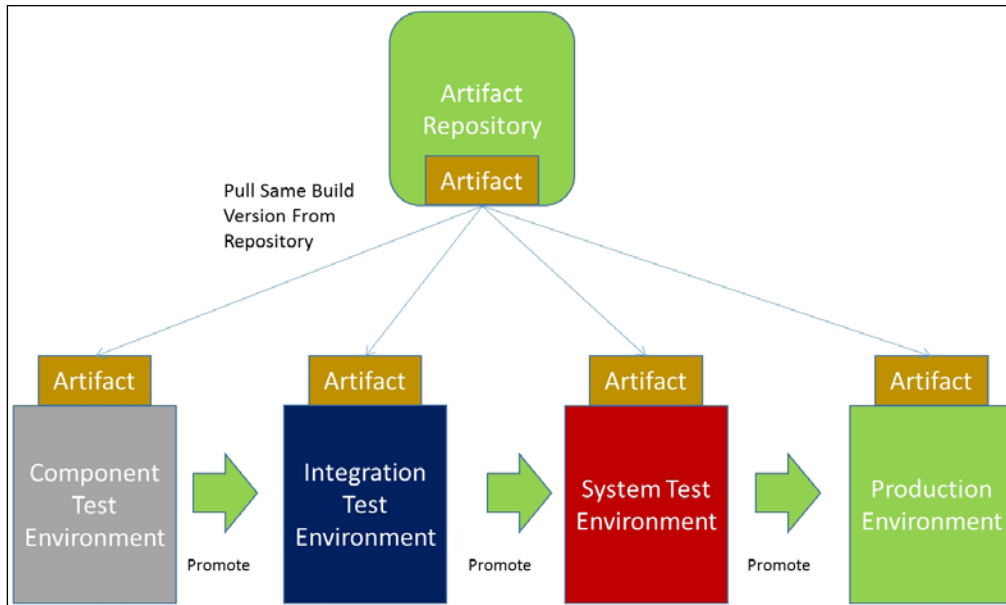
An important principle to remember when setting up continuous integration builds and packaging continuous integration artifacts, is that artifacts should be packaged once only, not every single time they need to be deployed.

This is important from a repeatability perspective and also reduces the time taken to deploy as a build process can be lengthy and take many minutes. When a build has been packaged, all tests and necessary validation have been carried out on the artifact as part of the continuous integration process, so there is no need to repeat this process again if no changes have occurred.

It is imperative that we ensure the exact same artifact is deployed to test environments before being promoted onto production; this means there will be no drift between environments. The same source code being packaged on a different build server may result in the version of Java being slightly different, or even something as simple as a different environment variable could mean the build binaries are compiled differently.

Maintain consistent deployment artifacts, always swearing by the principle of package once and deploy multiple times.

The standard of package once, deploy multiple times is illustrated following, where a single artifact is used to seed test and production environments:



Creating different build artifacts for each environment is a non-starter; release management best practices dictate that a build package and artifacts should include tokens so different snapshots of the same package are not required.

Build package tokens can then be transformed at deployment time. All environment specific information is held in a configuration file of some sort, normally called an **environment** file, which is used to populate the tokens at deployment time.

The following best practices should be adhered to when packaging continuous integration build artifacts:

- Artifacts should be packaged once and distributed many times
- Artifact packages should be packaged with tokenized configuration files
- Artifact package configuration files should be transformed at deployment time using an environment file
- Common files can be used to supplement environment files if deployment configuration is common to all environments to avoid repetition

The following popular configuration management tools have the ability to transform tokenized templates by utilizing configuration files. Each of these configuration files take on the role of the environment file:

- Puppet <https://puppet.com/>
- Chef <https://www.chef.io/chef/>
- Ansible <https://www.ansible.com/>
- Salt <https://saltstack.com/>

Taking Ansible as an example, in *Chapter 4, Configuring Network Devices Using Ansible*, we covered the concept of jinja2 templates. Jinja2 templates allow template files to be populated with tokens and these tokens are substituted with particular key value pairs at deployment time.

Ansible allows users to populate jinja2 templates to be populated with variables (tokens). Each `var` file can be configured so that it is unique to each environment. Environment files can be imported into playbooks and roles by inputting it as a command line argument. This will in turn transform the jinja2 templates at deployment time with the environment-specific information.

In the following example, we see an Ansible playbook `configure_env.yml` being executed, and a unique environment variable called `environment` needing to be set:

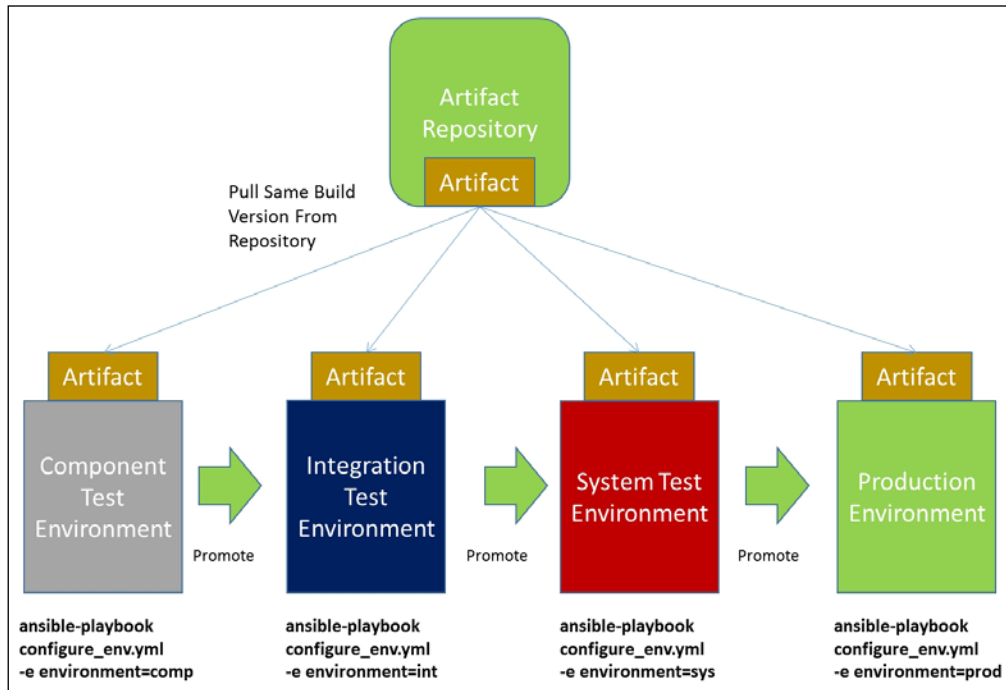
```
- name: Include vars
  include_vars: "../roles/networking/vars/{{ item }}.yml"
  with_items:
    - "common"
    - "{{ environment }}"
```

This will be imported into the playbook `configure_env.yml` so that a unique set of environment information is loaded for each environment.

Therefore, taking the component, integration, system test, and production environments as an example the following files would be loaded:

- `../roles/networking/vars/comp.yml`
- `../roles/networking/vars/int.yml`
- `../roles/networking/vars/sys.yml`
- `../roles/networking/vars/prod.yml`

For each unique environment, the deployment command differs only in the environment file that is loaded which will make the deployment environment specific:



Continuous Delivery and deployment overview

Continuous Delivery and deployment are a natural extension of the continuous integration process. Continuous Delivery and deployment create a consistent mechanism to deploy changes to production and create a conveyor belt delivering new features to customers or end users. So conceptually a conveyor belt is what continuous Delivery is all about, but in terms of actual process how is this achieved?

A continuous integration process will carry out the following high level steps:

- Commit
- Build (Compile/Version/Tag)

- Validate
- Package
- Push

Continuous Delivery and deployment take over once the artifact has been pushed to the artifact repository. Each and every build artifact created by a continuous integration process should be considered a release candidate, meaning that it can potentially be deployed to production if it passes all validations in the Continuous Delivery pipeline.

Like continuous integration, Continuous Delivery and deployment create a series of feedback loops to indicate if validation tests have failed on an environment.

A Continuous Delivery pipeline process will encapsulate the following high level steps at each stage of a deployment pipeline:

- Deploy (pull/tokenize/setup)
- Validate (test)
- Promote

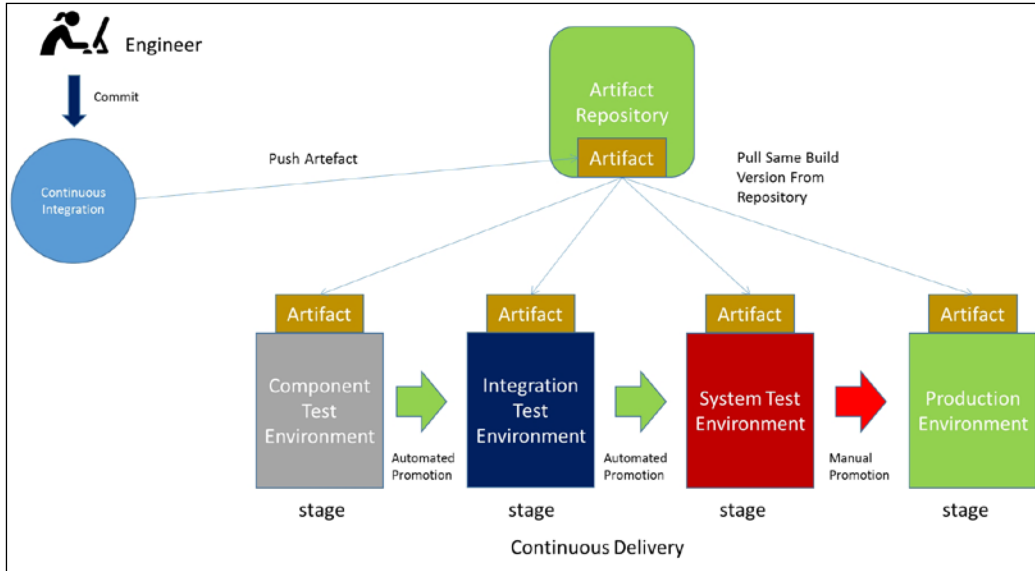
A stage in a deployment pipeline will contain a series of tests which will be used to help validate whether the application is functioning as required prior to it being released to production.

Each stage in the deployment pipeline will have a deployment step which will pull down the artifact from the **Artifact Repository** to the target server and execute the deployment steps. The deployment process will normally involve installing software or configuring a change to the state of the server. Configuration changes are typically governed by a configuration management tool such as Puppet, Chef, Ansible, or Salt.

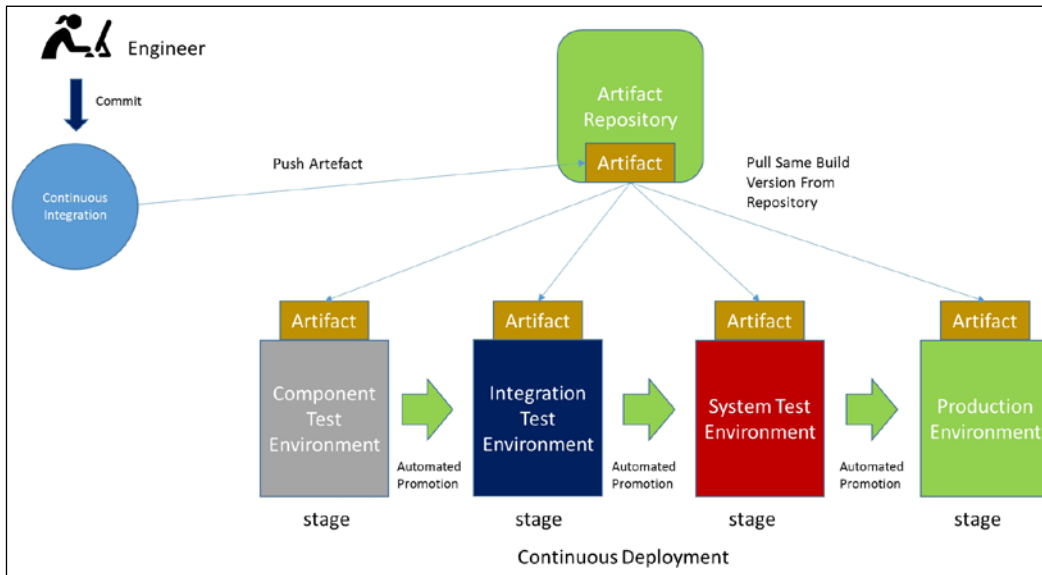
Once deployment is completed, a series of tests will be carried out in the environment to validate the deployment and also test the functionality of the application or change.

Continuous Delivery means that if validation tests pass on a test environment then the build artifact is automatically promoted to the next environment. The deployment, validation, and promotion steps are carried out again on the next environment in the same way as the previous environment. In the event of a failure, the release candidate will break and it will not be promoted to the next stage of the process.

When using Continuous Delivery this automatic promotion happens all the way to the environment prior to production as shown in the following diagram:



Continuous deployment on the other hand has no paused state before production and differs from Continuous Delivery in that it will automatically deploy to production:



So the only difference between Continuous Delivery and continuous deployment is the manual pause from promoting the build artifact to production.

The reason for implementing Continuous Delivery over continuous deployment is normally down to either governance or the maturity of testing.

When starting out, continuously deploying to production throughout the day can seem very daunting as it mandates that the deployment process is completely automated and that the validation and testing on each environment is mature enough to catch all known errors.

With continuous deployment, the trigger of a production deployment is a SCM commit, so it puts a lot of trust in the deployment system. This means it is desirable that the branching strategy is set up to pull all changes from the **trunk/mainline/master** branch and trigger the deployment pipeline. Having multiple different branches will complicate the deployment process so it is important to implement a branching strategy that minimizes repetition, and an explosion of the number of deployment pipelines that are required.

If implemented badly, continuous deployment can result in continuous downtime, so normally after setting up continuous integration businesses, teams should start with Continuous Delivery and aim to eventually move to a continuous deployment once processes have matured sufficiently.

As covered in *Chapter 3, Bringing DevOps to Network Operations*, cultural change is needed within the business to implement a Continuous Delivery model and it really is an all or nothing approach for it to work successfully.

As stated in *Chapter 8, Testing Network Changes*, manually updating environments can compromise the validity of tests so they should be avoided at all costs, every change should flow through SCM to downstream environments.

Continuous Delivery promotes automation and creation of test packs at every stage in the deployment pipeline, but it also allows a business to cherry-pick the release candidate that is finally deployed to production.

This means additional validation could be carried out manually during the imposed stop before production, in the absence of the desired level or test coverage for a build artifact. It also plays well with companies that are subject to regulatory requirements that may mean they only have a specified deployment window and they cannot deploy to production continuously.

Continuous Delivery means that regulated companies can still benefit from automated environments and tests, but the production deployment is just a button click to select the artifact, which has passed all aforementioned promotions and is deployed to production.

Deployment methodologies

When carrying out Continuous Delivery and deployment, there is no one size fits all deployment strategy. Configuration management tools such as Puppet, Chef, Ansible, and Salt have different approaches to deployment and use different approaches when keeping servers up to date.

The tool that is selected is not important, only the ideal workflow and processes to support delivering changes that are consistent, quick, and accurate.

Pull model

Tools such as Puppet and Chef adopt a centralized approach to configuration management, where they have a centralized server that acts as the brain for the deployment process.

In Puppet's case the centralized server are the Puppet Master and in Chef's case the centralized server is the Chef Server. This centralized server is a set of infrastructure provisioned to store server configuration according to the configuration management tool's reference architecture.

All updates to server configuration is pushed to the centralized server first and then subsequently pushed out to the corresponding servers using agents. These agents can either poll the centralized server for updates and apply them straight away or alternately wait for the Puppet Agent, or in Chef's case, the Chef Client to be invoked to start the convergence of configuration from the centralized server to the server containing the agent.

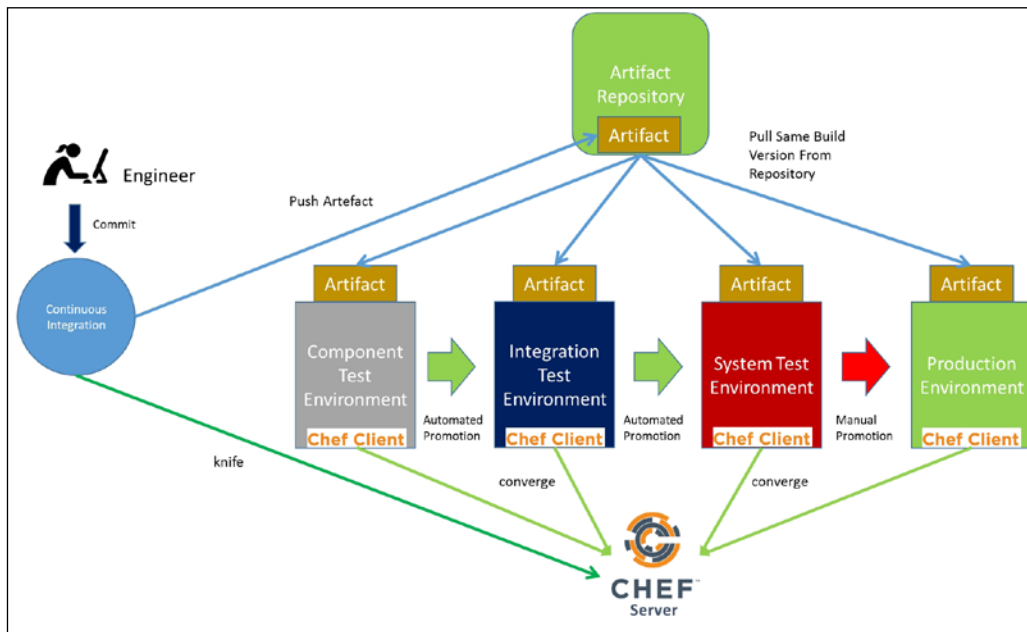
The overriding principle in a pull model is that the centralized server governs the state of the system and every change goes via the Puppet Master or Chef Server.

If any user logs onto a server and changes the state, then the next time that the state converges from the centralized server it could overwrite those manual changes when the agent runs (Puppet Agent or Chef Client) if that particular configuration is managed by the centralized server.

In this model, the centralized server will control all application versioning information and environment configuration.

An example of a pull model is shown in the following diagram. This shows Chef being used in a Continuous Delivery process:

- The continuous integration process creates a new build artifact which is pushed to the **Artifact Repository**
- Chef's command line client **knife** is invoked as a post-build action which updates the **Chef Server** with the new version of the application which is being deployed
- The deployment process is then triggered on **Component Test Environment** by running **Chef Client** which will trigger the **Chef Client** to check the state against the **Chef Server**
- The **Chef Client** in this case, sees a new application version is available based on the last **knife** update and as a result updates the environment to the new version of the application
- Finally, all validation and test steps are run prior to promoting it to the next stage of the deployment pipeline
- Convergence on the subsequent **Integration Test Environment** is only triggered if the **Component Test Environment** promotion is successful



Push model

Tools such as Ansible and Salt adopt a push model to configuration management, where they have a control host that is used to connect to servers using SSH and configure them.

Instead of using a centralized server, Ansible and Salt use a control host, which has a command-line client installed on the server. The control host is then used to push changes to servers via logging on to them using SSH either via password or alternatively, SSH keys.

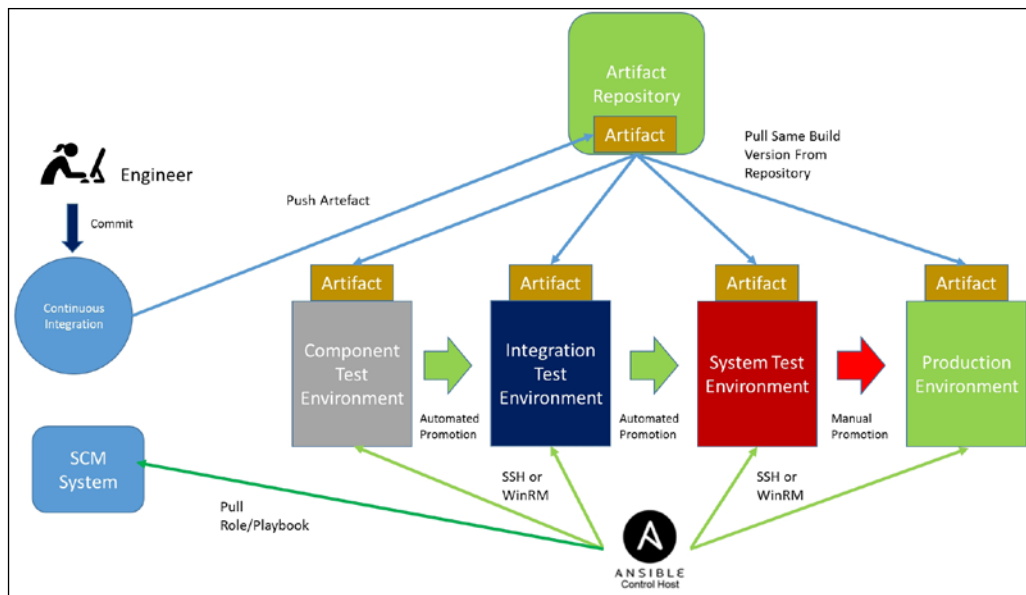
As Ansible and Salt are Python-based, they are agentless and can run on any Linux distribution, as Python is a pre-requisite for these servers. Windows machines are connected to and configured using WinRM.

All configuration management information is stored in SCM systems and pulled down to the control host, this configuration is then used to push updates out to servers.

The overriding principle in a push model is that changes are committed into SCM. The SCM server, rather than a centralized server, is the source of truth for state, configuration, and versioning.

An example of a push model follows. This shows Ansible being used in a Continuous Delivery process:

- The continuous integration process creates a new build artifact which is pushed to the **Artifact Repository**
- A new artifact being present in the **Artifact Repository** triggers the deployment process and the Ansible playbook/role is downloaded from the **SCM System** to the **Ansible Control Host**
- The deployment process is then triggered on **Component Test Environment** and Ansible is executed against all servers that are present in the targeted inventory
- Finally, all validation and test steps are run prior to promoting it to the next stage of the deployment pipeline
- Ansible is only executed on the subsequent Integration Test Environment if the Component Test Environment promotion is successful



When to choose pull or push

When selecting a pull or push method of configuration management, it is down to preference and should be selected based on the approach to infrastructure.

Pull models are popular when dealing with server estates that have long-lived infrastructure. It lends itself well to patching a whole estate of servers to keep on top of compliance. Pull models, as they have a centralized server with the current state, means that if configuration is removed from a server, then the centralized server will register that a delete is required. Push models only understand the new desired state and don't take into account the previous state due to the lack of convergence. So if some configuration is removed from a playbook for example, it won't be automatically cleaned up when the next deployment occurs.

The drawbacks of a pull approach are the requirement to maintain the infrastructure for the centralized server which can be somewhat large, and as it is agent-based, agent versions also need to be maintained.

Push models align themselves well to orchestration and updating large amounts of servers. They are popular when using immutable infrastructure as the old state of the server is not important. This means that only the current desired state is relevant, so it is not necessary to clean-up deleted configuration as servers will be deployed at every deployment.

A pull model with immutable infrastructure wouldn't really make sense as the boxes would only converge once and then be destroyed, so the overhead of running large centralized servers to take care of convergence is wasteful.

Packaging deployment artifacts

Using configuration management tooling just to deploy applications is not enough; Continuous Delivery and deployment are only as quick as its slowest component. So having to wait for manual network or infrastructure changes is not an option; all components need to be built, versioned, and have their deployment automated.

When looking at building new environments from scratch, multiple deployment artifacts need to be used to build an environment; application code is just one piece of the jigsaw.

The following dependencies are required to build a redundant environment:

- Application
- Infrastructure (base operating systems and virtual or physical servers)
- Networking
- Load balancing
- Deployment scripts (configuration management)

Not versioning all these components together means that true rollback is not available as components may break if an application is rolled back and the network has moved forward in terms of state.

Ideally application code, infrastructure, networking, load balancing, and deployment scripts should all be versioned and tested together as one entity. So if rollback is required then operators can simply roll-back to the last known package which has tried and tested versions of the application code, infrastructure, networking, load balancing, and deployment scripts that were known to work together.

One option is to have a single repository that versions all dependencies in that one repository. This can be inflexible when dealing with large numbers of applications and can result in repetition of configuration.

Another way to version all components is via continuous integration builds, each of the components can have their own continuous integration build to version the individual components and a unique repository.

Applications will be a packaged entity which may be an RPM file on Red Hat Linux, APT file on Ubuntu, or a NuGet package on Windows.

Infrastructure will be provisioned using cloud provider APIs such as OpenStack, Microsoft Azure, Google Cloud, or AWS, so the desired number of servers will need to be specified using a version controlled inventory file.

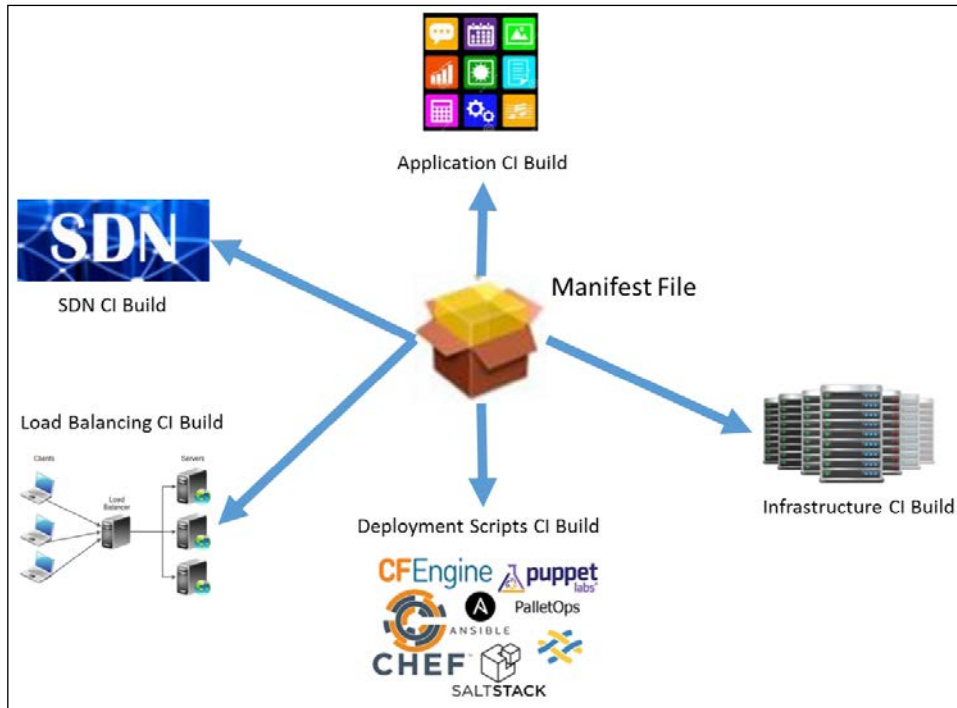
The base operating system images can be created using tooling such as Packer or OpenStack Disk Image Builder and uploaded to a cloud provider's image registry.

As covered in *Chapter 4, Configuring Network Devices Using Ansible*, *Chapter 5, Configuring Load Balancers Using Ansible*, and *Chapter 6, Configuring SDN Controllers Using Ansible*, network configuration, when utilizing Ansible, normally takes the form of `var` files which describe the desired state of the system.

When using an SDN controller, the subnet ranges and ACL firewall rules can be described in these `var` files and utilize modules scheduled in specific orders to apply them at deployment time. In a similar vein, the load balancing configuration object model can be stored in Ansible `var` files to set up load balancing.

Each of these repositories should be tagged as part of the continuous integration build and a supplementary package build can then be created for each application. This package build is used to roll-up all the dependencies and version them together using a manifest file.

The continuous integration builds that contribute to the manifest file are shown in the following diagram:



A manifest file can take the form of a simple key value pair file or a JSON file. The format of the file is not important, recording the latest tagged version of each continuous integration build is integral to the process.

At deployment time, a new packaged manifest should be used as the trigger for the deployment pipeline. The first step of the deployment pipeline will pull down the manifest file from the artifact repository and it can then be read for version information.

All versions of the repositories present in the manifest file can then be pulled down to the Ansible control server and used to deploy the desired application version along with the desired state to the infrastructure, network, and load balancer required for each environment.

Roll-back would involve passing the previous version of the manifest file to the deployment process which would then revert to the last tried and tested versions of the application code, infrastructure, networking, load balancing, and deployment scripts that were known to work together.

Deployment pipeline tooling

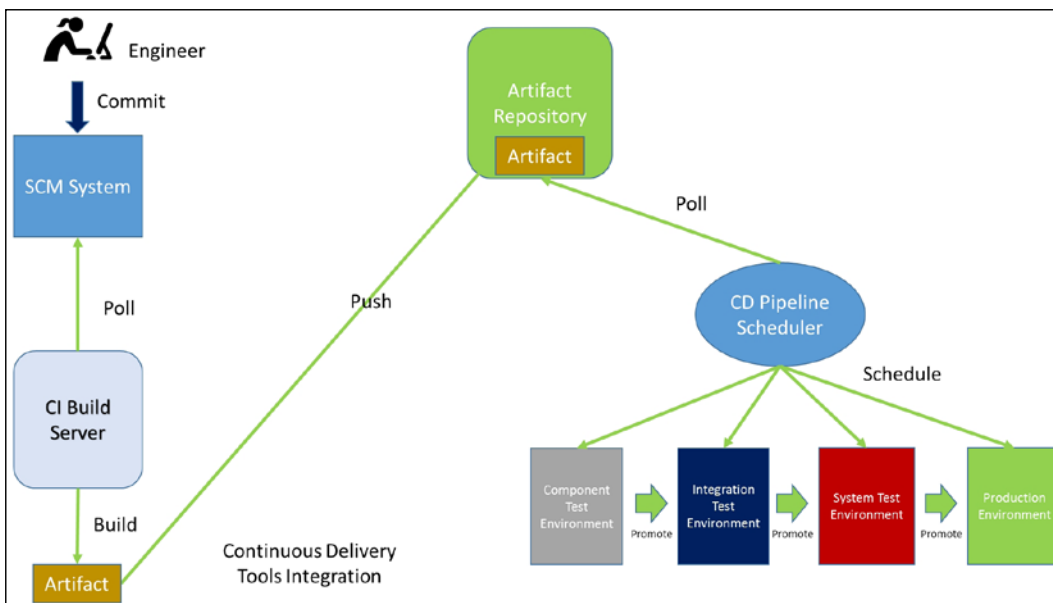
Deployment pipelines involve chaining different tools together to create Continuous Delivery processes.

Being able to track the process flow through the Continuous Delivery tooling is integral, as it is important to be able to visualize the pipeline process, so it is easy for operators to follow.

Having visibility of a process makes debugging the process easy if errors occur, which may happen as errors will occur in any process and are inevitable. The whole point of the Continuous Delivery pipeline, aside from automating delivery of changes to environments, is to provide feedback loops. So if a pipeline is not easy to follow and debug, it has failed one of its main objectives.

Building automatic clean-up into pipelines should be implemented if possible, so if a failure occurs mid-deployment then changes can be reverted back to the last known good state without the need for manual intervention.

At a high level, the following tooling is required when creating a deployment pipeline for Continuous Delivery which includes a **SCM System**, **CI Build Server**, **Artifact Repository**, and **CD Pipeline Scheduler**:



In *Chapter 7, Using Continuous Integration Builds for Network Configuration*, and *Chapter 8, Unit Testing Network Changes*, we covered the importance of the SCM System and CI Build Server in continuous integration and testing. In this chapter we will focus on the tooling required for the deployment process which includes the Artifact Repository and CD Pipeline Scheduler that is used to schedule configuration management tooling.

Artifact repositories

Artifact repositories are a key component in any deployment pipeline; they can be used to host a multitude of different repositories or even just hold generic artifacts.

Platform golden images in ISO, AMI, VMDK, and QCOW format can be stored and versioned in artifact repositories and used as the source for image registries for cloud providers such as AWS, Google Cloud, Microsoft Azure, and OpenStack.

Manifest files can also be held in a release repository to govern the roll-forward and roll-back of application, infrastructure, networking, and load balancing requirements.

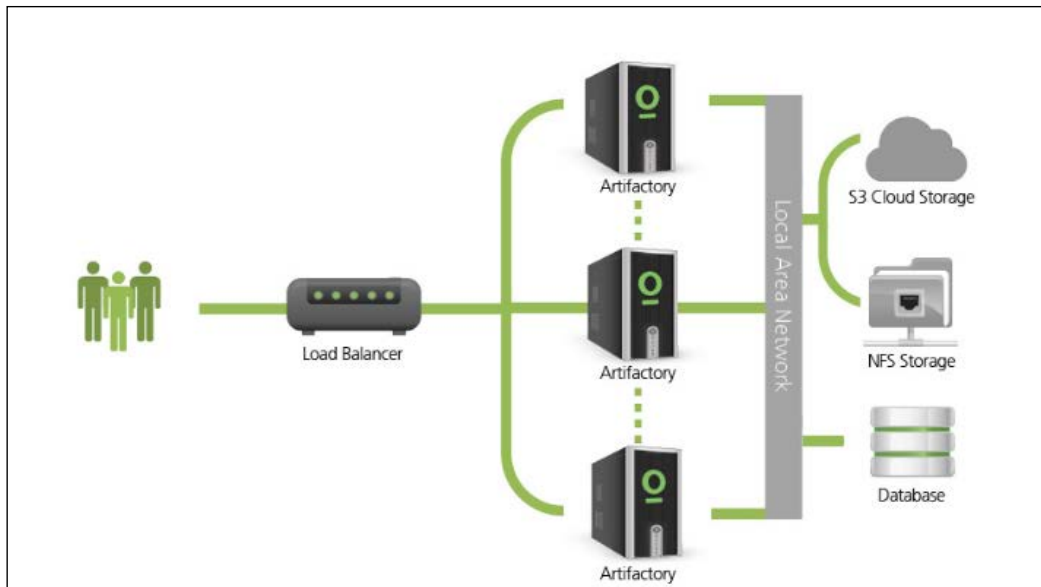
Artifactory

Artifactory from JFrog is one of the most popular artifact repositories on the market today and provides access to repositories via an NFS-based shared storage solution. Artifactory is bundled with the Apache Tomcat web server as part of the installer bundle and can be hosted on Linux or Windows.

In terms of load balancing, Artifactory can be set up in a highly available, three tier cluster for redundancy. Artifactory can use a wide variety of load balancers such as Nginx or HAProxy as well as proprietary load balancers such as Citrix NetScaler, F5 Big-IP, or Avi Networks.

Artifactory is backed by a MySQL or Postgres database and requires an NFS file-system or Amazon S3 storage to store artifacts that are made available to each of Artifactory's three HA nodes.

The architectural overview of **Artifactory** is shown in the following diagram:



Artifactory supports numerous different repository types, some of which are shown here, so it can host multiple different repositories for delivery teams depending on the applications that they are developing:

- **Maven** <https://www.jfrog.com/confluence/display/RTF/Maven+Repository>
- **Ivy** <https://www.jfrog.com/confluence/display/RTF/Working+with+Ivy>
- **Gradle** <https://www.jfrog.com/confluence/display/RTF/Gradle+Artifactory+Plugin>
- **Git LFS** <https://www.jfrog.com/confluence/display/RTF/Git+LFS+Repositories>
- **NPM** <https://www.jfrog.com/confluence/display/RTF/Npm+Registry>
- **NuGet** <https://www.jfrog.com/confluence/display/RTF/NuGet+Repositories>
- **PyPi** <https://www.jfrog.com/confluence/display/RTF/PyPI+Repositories>
- **Bower** <https://www.jfrog.com/confluence/display/RTF/Bower+Repositories>
- **YUM** <https://www.jfrog.com/confluence/display/RTF/YUM+Repositories>

- Vagrant <https://www.jfrog.com/confluence/display/RTF/Vagrant+Repositories>
- Docker <https://www.jfrog.com/confluence/display/RTF/Docker+Registry>
- Debian <https://www.jfrog.com/confluence/display/RTF/Debian+Repositories>
- SBT <https://www.jfrog.com/confluence/display/RTF/SBT+Repositories>
- Generic <https://www.jfrog.com/confluence/display/RTF/Configuring+Repositories>

This means Artifactory can be used as the single repository end-point for Continuous Delivery pipelines. Artifactory has recently introduced support for Vagrant boxes and Docker registry so it can be used to store Vagrant test environments, which could be used to store network operating systems or containers. This illustrates some of the features available from market-leading artifact repositories.

CD pipeline scheduler

While the job of the artifact repository is relatively straightforward, but no less important, choosing the correct Continuous Delivery pipeline tool is much more difficult.

There is a wide array of options available such as:

- IBM Urban Code Deploy <https://developer.ibm.com/urbancode/products/urbancode-deploy/>
- Electric Flow Deploy <http://electric-cloud.com/products/electricflow/deploy-automation/>
- Jenkins <https://jenkins.io/>
- Thoughtworks Go <https://www.go.cd/>
- XL Deploy <https://xebialabs.com/products/xl-deploy>

But before picking a tool, the process being implemented needs to be considered. So what are the main aims of a Continuous Delivery pipeline?

A good Continuous Delivery pipeline should meet the following goals:

- Trigger deployments based on new artifacts being available in artifact repository
- Schedule command lines
- Render a pipeline view
- Break tasks into stages
- Provide good log output
- Feedback pass or failures
- Integration with testing

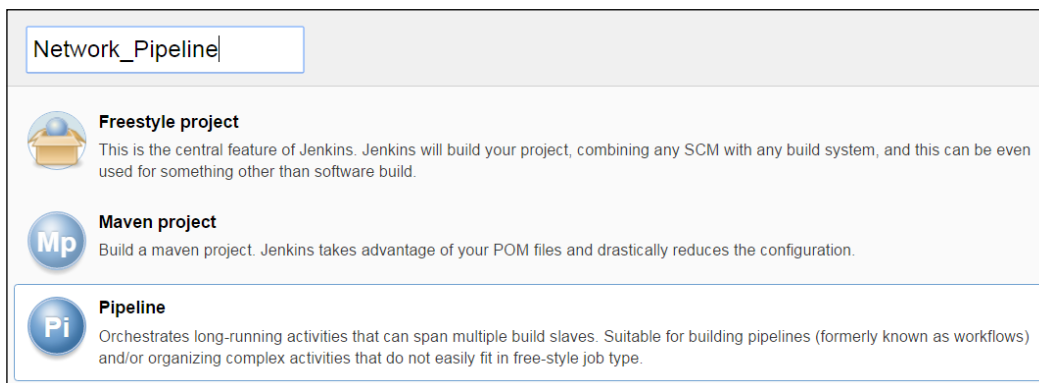
All of these points need to be considered when selecting tooling so we will cherry-pick one of the most popular Continuous Delivery pipeline scheduling tools Jenkins, and look at ways in which it schedules pipelines.

Jenkins

Jenkins was primarily a continuous integration build server when it was conceived. Jenkins has a pluggable framework that means that it is often customized to carry out deployments, with plugins such as the multi-job plugin built to allow it to schedule pipelines.

However, as of the Jenkins 2.x release, Jenkins now makes pipelines a core feature component of its distribution rather than depending on plugins to cater for its deployment capabilities.

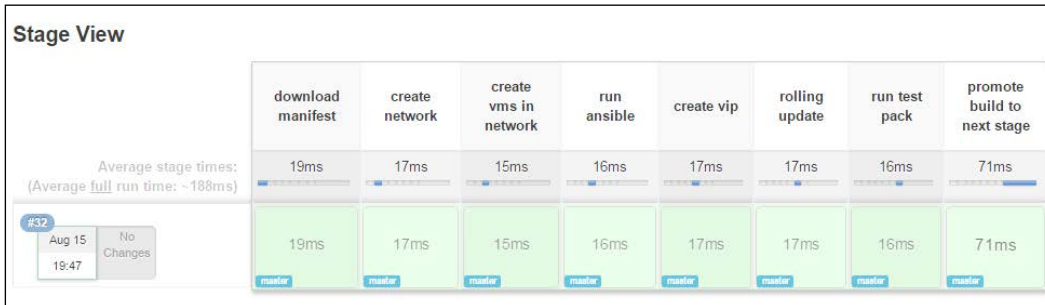
In the following example, the Jenkins Pipeline job type can be seen:



Using the Jenkins Pipeline job type, users can specify pipelines using a Pipeline Script, declaring each stage of the pipeline. In this instance, the echo command has been used to spoof each pipeline stage to show how a Pipeline script may look:

```
Definition Pipeline script
Script
1 node {
2   stage 'download manifest'
3   echo 'downloaded manifest'
4   stage 'create network'
5   echo 'created network'
6   stage 'create vms in network'
7   echo 'created vms in network'
8   stage 'run ansible'
9   echo 'ran ansible'
10  stage 'create vip'
11  echo 'created vip'
12  stage 'rolling update'
13  echo 'rolled new boxes into service and old ones out'
14  stage 'run test pack'
15  echo 'ran test pack'
16  stage 'promote build to next stage'
```

The visual display of the pipeline from the pipeline script is shown in the **Stage View**. The **Stage View** shows the eight stages the pipeline went through in order to deploy the networking, virtual machines, application, and load balancer configuration:

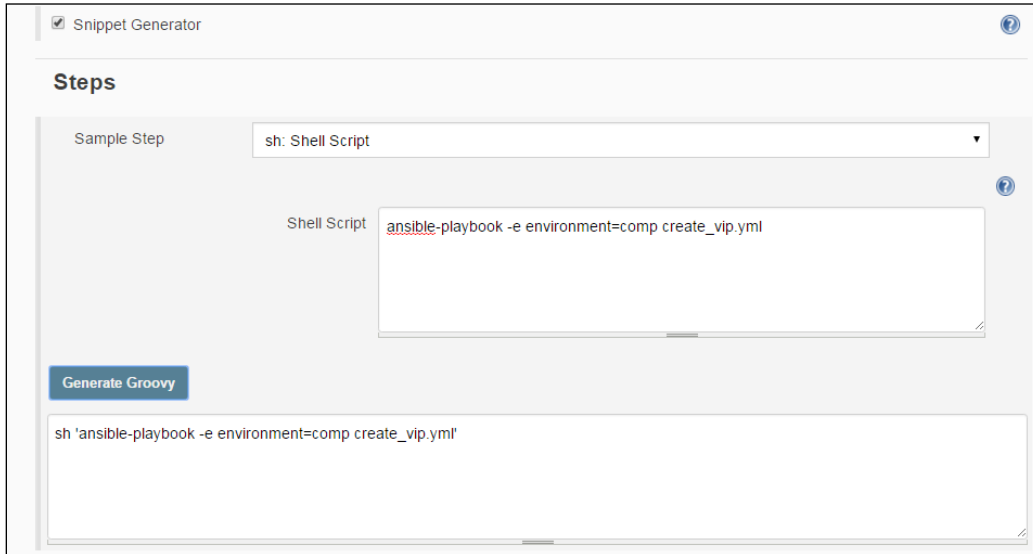


Logging for each stage is shown clearly in the Jenkins console logs which allow users of the tool to see feedback on successful and unsuccessful console logs:

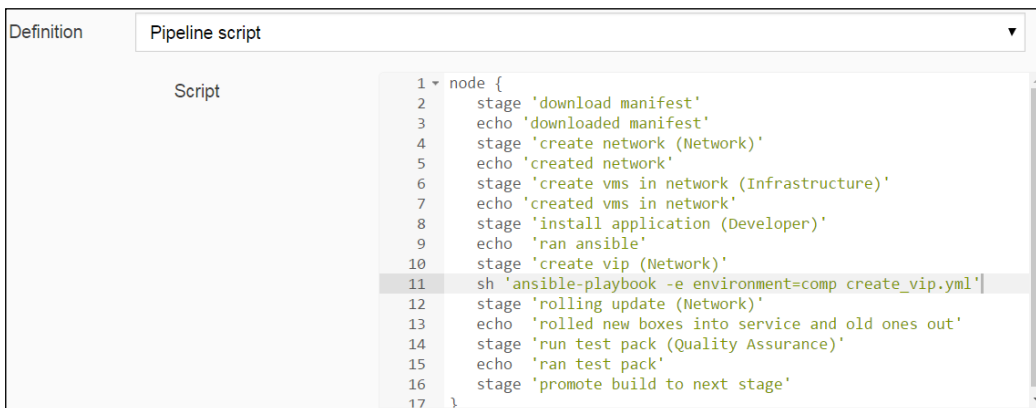
```
[Pipeline] node {
[Pipeline] stage (download manifest)
Entering stage download manifest
Proceeding
[Pipeline] echo
downloaded manifest
[Pipeline] stage (create network)
Entering stage create network
Proceeding
[Pipeline] echo
created network
[Pipeline] stage (create vms in network)
Entering stage create vms in network
Proceeding
[Pipeline] echo
created vms in network
[Pipeline] stage (run ansible)
Entering stage run ansible
Proceeding
[Pipeline] echo
ran ansible
```

When putting valid commands into `Pipeline` script, as opposed to simulating echoes as we have in the above example, Jenkins allows users to use a groovy snippet generator to translate any steps to Pipeline Script format.

In this instance, a shell command is required to execute the Ansible playbook to create `create_vip.yml` on the component test environment, so the snippet generator is used to create it:



This snippet command can then be pasted into the `create_vip.yml` stage that was created on the Pipeline script:



The output of the job configuration is a Jenkins file that can be stored in SCM to version control the deployment pipeline changes.

Deploying network changes with deployment pipelines

When carrying out Continuous Delivery or deployment, it is essential to incorporate network changes. Network teams need to contribute major pieces of the deployment pipeline.

As the CD Pipeline scheduler allows different stages to be specified in the deployment pipeline, it gives great flexibility and allows all teams to contribute pieces, forming a true collaborative DevOps model.

Sometimes a concern from network teams is that developers should not have the necessary access to all network devices as they are not experts. Truth be told, developers don't want access to network devices, they instead want a quick way of pushing out their changes where they are not impeded by having to wait on network changes being applied.

Network self-service

Allowing developers the ability to self-service their own network changes is very important, otherwise the network team becomes the bottleneck for the Continuous Delivery process.

So providing development teams with, say, a hardened Ansible playbook to create everyday network functions will undoubtedly help alleviate developer pain and make deployment of new network changes a self-service function.

Developers can use a playbook that incorporates all the best practices of the network team to apply any network changes. This is following the model where developers can utilize a playbook provided by the infrastructure team to spin up new virtual machines and register their DNS entries with the IPAM solution.

Steps in a deployment pipeline

When creating deployment pipelines, it is important to break up each function into a granular set of steps. This means if any step fails it can be easily rolled back. Understanding the deployment pipeline visually is also important as breaking down complex operations into small steps makes debugging failures less daunting too.

A modern application deployment pipeline will provision new environments by carrying out the following high level steps every single deployment:

1. Download manifest
2. Create network
3. Create VMs in network
4. Install application
5. Create VIP
6. Rolling update
7. Run test pack
8. Promote to next phase

The first stage of the pipeline is the trigger for a new deployment to the first test environment. In this case, the detection of a new manifest file artifact.

The manifest artifact will be downloaded to the CD Pipeline scheduler and parsed. The Ansible `var` file structure will be assembled from SCM using the manifest versions.

Once assembled, the network needs to be provisioned. An A or B network will be created depending on the release and the necessary Ingress and Egress ACL rules will be applied to the network.

Virtual machines will then be booted into the newly-provisioned network and tagged with their metadata profile stating the software that needs to be installed on them.

Ansible dynamic inventory is run to pull back the new virtual machines that were just created, Ansible reads the profile metadata from the virtual machines. metadata tags and Ansible installs the required role on the new cluster of virtual machines depending on what profile is specified.

A VIP is created on the load balancer if it doesn't already exist and its load balancing policies are applied. Boxes are then rolled into service on the new VIP and old boxes are rolled out of service. The new boxes are smoke tested to make sure they are operating as expected before the previous release is destroyed.

A full quality assurance test pack is then executed and the manifest artifact is then promoted to the next stage if successful.


Each of these steps will be repeated all the way up to production. In a Continuous Delivery model, the production deployment will be a manual button press to trigger the pipeline, where in Continuous Delivery pipeline will automatically trigger if all quality gates pass.

Incorporating configuration management tooling

When utilizing a CD scheduler such as Jenkins, its agents, known as slaves, can be used to install Ansible on them and they become the Ansible Control Host for the deployment.

Each stage in the deployment pipeline can be a small modular Ansible playbook that allows developers to self-serve their network needs. These playbooks can be created by the network team and continuously improved over time.

So the Jenkins Pipeline script would resemble the following, with a unique playbook for each stage:



The screenshot shows a Jenkins Pipeline configuration window titled "Pipeline". Under the "Definition" tab, a dropdown menu is set to "Pipeline script". Below this, a "Script" field contains a Groovy pipeline script. The script is as follows:

```

1 node {
2     stage 'download manifest'
3     sh 'ansible-playbook download_manifest.yml'
4     stage 'create network (Network)'
5     sh 'ansible-playbook -e environment=comp create_network.yml'
6     stage 'create vms in network (Infrastructure)'
7     sh 'ansible-playbook -i inventories/inventory -l qa -e environment=comp create_vms.yml'
8     stage 'install application (Developer)'
9     sh 'ansible-playbook -i inventories/openstack.py -l qa -e environment=comp install_application.yml'
10    stage 'create vip (Network)'
11    sh 'ansible-playbook -e environment=comp create_vip.yml'
12    stage 'rolling update (Network)'
13    sh 'ansible-playbook -i inventories/openstack.py -l qa -e environment=comp rolling_update.yml'
14    stage 'run test pack (Quality Assurance)'
15    sh 'ansible-playbook -e environment=comp run_selenium.yml'
16    stage 'promote build to next stage'
17 }
18

```

The steps applied on each test environment should be consistent with production and all steps should be carried out by a service account for the pipeline.

Each and every environment should be built from source control by implementing immutable infrastructure and networking. This is so that the desired state is always what is specified in the manifest file's associated repositories.

The Ansible `var` files that feed each playbook can be filled in by the development teams in order to set firewall policies or load balancing policies.

These `var` files are versioned by the associated continuous integration builds for the SDN or load balancing configuration. Each network-related CI build then rolls up into a new manifest file when an application continuous integration build is triggered. The generation of a new manifest file triggers the first step in the deployment pipeline.

Network teams' role in Continuous Delivery pipelines

When analyzing the steps that are executed by a deployment pipeline, if we look at which teams would have the necessary permissions to carry out each pipeline stage manually, it becomes very apparent the importance of integrating networking into the Continuous Delivery processes.

Out of the eight high level stages to deploy an application, three of them are integrating with the network when executing **create network**, **create vip**, and **rolling update** as shown here:

download manifest	create network (Network)	create vms in network (Infrastructure)	install application (Developer)	create vip (Network)	rolling update (Network)	run test pack (Quality Assurance)	promote build to next stage
14ms	14ms	16ms	16ms	15ms	17ms	15ms	63ms
14ms	14ms	16ms	16ms	15ms	17ms	15ms	63ms
master	master	master	master	master	master	master	master

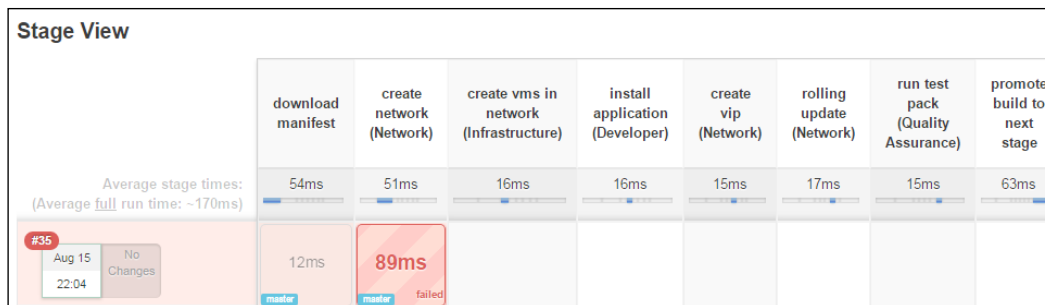
This shows that if network operations were not part of the deployment pipeline then true Continuous Delivery would not be achievable.

Failing fast and feedback loops

One of the key objectives of creating Continuous Delivery pipelines is creating feedback loops which fail fast and create a radiator view for developers. However, with Continuous Delivery moving into continuous operations space, as it now incorporates infrastructure, networking, and quality assurance, all teams need to be mindful of failures and react accordingly.

When pipeline stages fail, it is important to incorporate automated clean-up every time there is a failure, this leaves the pipeline in a good state so the next pipeline is not impeded. Any break in the process means that changes cannot reach production.

So although it may be a test environment that is breaking, it is now blocking potential fixes being deployed to production. If a failure occurs, the pipeline should also halt the whole process and not proceed to the next stage as shown below:



Ansible block rescue functionality is very useful when dealing with failed pipeline stages and clean-up, providing a try and catch-like feature for playbooks and roles.

Testing should also be incorporated into the deployment pipeline so if the run test stage of the pipeline fails, then there is a history of why the tests failed that can be audited. Pipelines also help provide a full history of changes that have been applied to the environment. Although triggered by a service account, the user that committed the change in source control should take ownership for each change.

Summary

In this chapter, we looked at integrating network changes into deployment pipelines so that network teams can contribute to the Continuous Delivery process. We then discussed the difference between Continuous Delivery and deployment.

We then looked at how package management is crucial for wrapping development, infrastructure, quality assurance, and network changes together as part of deployment pipelines. We also illustrated some of the market-leading artifact repositories and CD pipeline schedulers using Artifactory and Jenkins as examples.

Finally, we looked at best practices that should be adopted when setting up deployment pipelines within the remits of Continuous Delivery and deployment. We then focused on ways network teams could contribute to deployment pipelines by providing self-service deployment scripts to developers, so they keep the overall process quick, lean, and automated.

After reading this chapter, you should now understand why that applications should be compiled only once and stored in an artifact repository, and the same binaries should be deployed to multiple environments so the deployment process is consistent.

The chapter also focused on the differences between pull-based tools, such as Chef and Puppet, and tools such as Ansible and Salt that utilize a push model for configuration management.

Key takeaways should also include how to utilize Artifactory as an artifact repository to store numerous types of build artifacts, and ways in which manifest files can be generated using continuous integration to version code, infrastructure, networking, and load balancing.

Readers should learn all the necessary steps in a Continuous Delivery pipeline, how to set up a deployment pipeline using Jenkins 2.x, and the importance of integrating networking in the Continuous Delivery model.

In the next chapter, we will focus on containers and look at the impact they have had on networking and network operations. We will look at some of the different orchestration options that can be used such as Docker and Kubernetes.

10

The Impact of Containers on Networking

No modern IT book would be complete without a chapter on containers. In this chapter, we will look at the history of containers and the options currently available to deploy them. This chapter will look at the changes required to support running containers from a networking perspective. We will then focus on some of the technologies used to package containers, and how they can be incorporated into a Continuous Delivery process. Finally, we will focus on some of the orchestration tools that are being used to deploy containers.

In this chapter, the following topics will be covered:

- Overview of containers
- Packaging containers
- Container orchestration tools
- How containers fit into continuous integration and delivery

Overview of containers

There has been a lot of hype about containers in the IT industry of late; you could be forgiven for thinking that containers alone will solve every application deployment problem possible. There have been a lot of marketing campaigns from vendors stating that implementing containers will make a business more agile or that they mean a business is implementing *DevOps* simply by deploying their applications in containers. This is undoubtedly the case if you listen to software vendors promoting their container technology or container orchestration software.

Containers are not a new concept, though. Far from it: Solaris 10 introduced the concept of Solaris Zones as far back as 2005, which allowed users to segregate the operating system into different components and run isolated processes. Modern technologies such as **Docker** or **Rocket** provide a container workflow that allows users to package and deploy containers.

However, like all infrastructure concepts, containers are simply facilitators of process, and implementing containers as a standalone initiative for the wrong reasons will likely bring no business value to a company. It seems it has become almost mandatory for large software vendors to have a container-based solution as part of their portfolio, given their recent popularity.

Containers, like all tools, can be very beneficial for certain use cases. It is important when considering containers to consider the benefits that they bring to microservice architectures. It is fair to say that containers have been seen by some **Platform as a Service (PaaS)** companies as being the bridge between development and operations.

Container technologies have allowed developers to package their applications in containers in a consistent way, while at the same time describing the way in which they wish to run their microservice application in production using PaaS technology. This construct can be understood by development and operations staff, as they are both familiar with the same container technology and constructs they use to deploy applications. This means that the container that is deployed on a development workstation will behave in the same way as it would on a production system.

This has allowed developers to define their application topology and load balancing requirements more consistently, so that they are deployed identically to test and production environments using a common suite of tooling.

Famous success stories such as Netflix have shown that containerizing their whole microservice architecture is possible and can be a success. With the rise in popularity of microservice applications, a common requirement is to package and deploy a microservice application across multiple hybrid clouds. This gives organizations real choice over which private or public cloud provider they use.

In microservice architectures, cloud-native microservice applications can be scaled up or down quickly to deal with busy or quiet periods for a business. When using a public cloud, it is desirable to only utilize what is required, which can often mean that microservices can be scaled up and scaled down throughout the day to save running costs.

Elastic scaling based on utilization is a common use case when deploying cloud-native microservices so that microservices can scale up and down based on reading data from their monitoring systems or from the cloud provider.

Microservices have followed the lead of service-oriented architectures and can be seen as the modern implementation of this concept of **service-oriented architectures (SOA)**. Microservices such as SOA allow multiple different components to communicate via a network of services and common set of protocols. Microservices aim to decouple services from one another into specific functions, so they can be tested in isolation and joined together to create the overall system and, as illustrated, scaled up or down as required.

When using microservice architectures, instead of having to deploy the whole system each time, different component versions can be deployed independently of each other without causing system downtime.

Containers in some ways can be seen as the perfect solution for microservice applications as they can be used to carry out specific functions in isolation. Each microservice application can be deployed within the constructs of an individual container and networked together to provide an overall service to the end user.

Containers already natively run on any Linux operating system and are lightweight by nature, meaning they can be deployed, maintained, and updated easily when utilizing popular container technology such as:

- Docker (<https://www.docker.com/>)
- Google Kubernetes (<http://kubernetes.io/>)
- Apache Mesos (<http://mesos.apache.org/>)
- IBM Bluemix (<http://www.ibm.com/cloud-computing/bluemix/containers/>)
- Rackspace Catrina (<http://thenewstack.io/rackspace-carina-bare-metal-caas-based-openstack/>)
- CoreOS Rocket (<https://coreos.com/blog/rocket/>)
- Oracle Solaris Zones (https://docs.oracle.com/cd/E18440_01/doc.111/e18415/chapter_zones.htm#OPCUG426)

- Microsoft Azure Nano Server (<https://technet.microsoft.com/en-us/windows-server-docs/get-started/getting-started-with-nano-server>)
- VMware Photon (<http://blogs.vmware.com/cloudnative/introducing-photon/>)

Containerization in essence is virtualizing processes on the operating system and isolating them from one another into manageable components. Container orchestration technologies then create network interfaces to allow multiple containers to be connected to each other across the operating systems, or in more complex scenarios, create full overlay networks to connect containers running on multiple physical or virtual servers using programmatic APIs and key-value stores for service discovery.

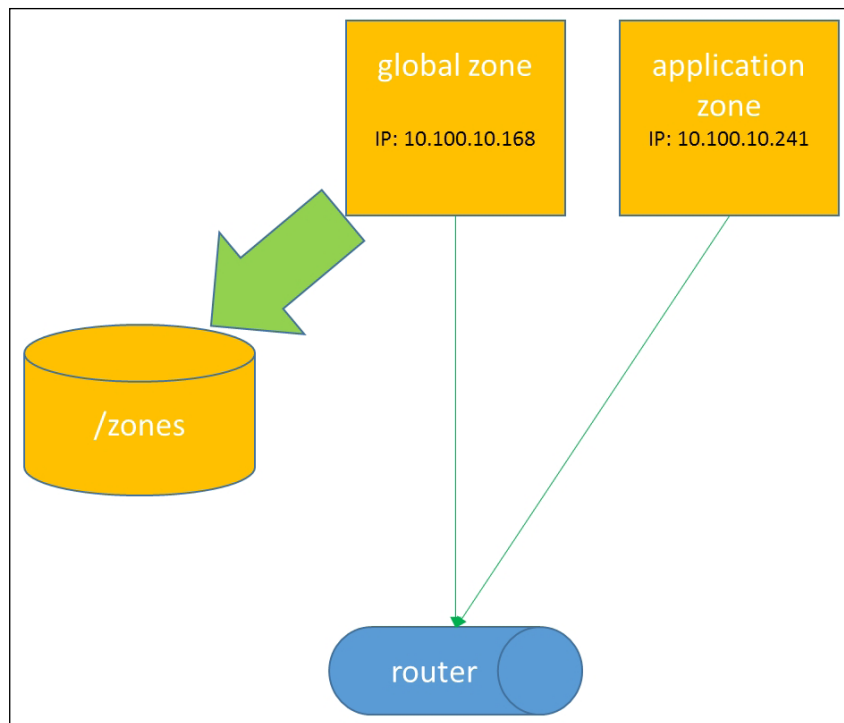
Solaris Zones

In 2005, Solaris introduced the notion of **Solaris Zones**, and from it came the concept of containment. After a user logged in to a fresh Solaris operating system, they would find themselves in a global Solaris Zone.

Solaris then gave users the option to create new zones, configure them, install the packages to run them, and finally, boot them so they could be used. This allowed each isolated zone to be used as a contained segment within the confines of a single Solaris operating system.

Solaris allowed zones to run as a completely isolated set of processes, all from the default global zone in terms of permissions, disk, and network configuration. Different persistent storage or raw devices could be exported to the zones and mounted to make external file systems accessible to a zone. This meant that multiple different applications could run within their own unique zone and communicate with external shared storage.

In terms of networking, the global Solaris Zone would have an IP address and be connected to the default router. All new zones would have their own unique IP address on the same subnet using the same default router. Each zone could even have their own unique DNS entry if required. The networking setup for Solaris Zones is shown in the following figure, with two zones connected to the router by accessing the network configuration on the `/zones` file system:



Linux namespaces

A **Linux namespace** creates an abstraction layer for a system process and changes to that system process only affect other processes in the same namespace. Linux namespaces can be used to isolate processes on the Linux operating system; by default, when a Linux operating system is booted, all resources run under the default namespace so have the ability to view all the processes that are running.

The namespace API has the following system calls:

- `clone`
- `setns`
- `unshare`

The `clone` system call creates a new process and links all specified processes to it; the `setns` system call, on the other hand, is used to join namespaces together, and the `unshare` system call moves a process out of a namespace to a new namespace.

The following namespaces are available on Linux:

- Mounts
- Process ID
- Interprocess communication
- UTS
- Network
- User

The mounts namespace is used to isolate the Linux operating system's file system so specific mount points are only seen by a certain group of processes belonging to the same namespace. This allows different processes to have access to different mount points, depending on what namespace they are part of, which can be used to secure specific files.

The **Process ID (PID)** namespace allows the reuse of PID processes on a Linux machine as each set of PIDs is unique to a namespace. This allows containers to be migrated between hosts while keeping the same PIDs, so the operation does not interrupt the container. This also allows each container to have its own unique `init` process and makes containers extremely portable.

The **interprocess communication (IPC)** namespace is used to isolate certain specific resources, such as system objects and message queues between processes.

The UTS namespace allows containers to have their own domain and host name; this is very useful when using containers, as orchestration scripts can target specific host names as opposed to IPs.

The network namespace creates a layer of isolation around network resources such as the IP space, IP tables, and routing tables. This means that each container can have its own unique networking rules.

The user namespace is used to manage user permissions to namespaces.

So, from a networking perspective, namespaces allow multiple different routing tables to coexist on the same Linux operating system, as they have complete process isolation. This means each container can have its own unique networking rules applied if desired.

Linux control groups

The use of **control groups (cgroups)** allows users to control Linux operating system resources that are part of a namespace. The following cgroups can be used to control Linux resources:

- CPU
- Memory
- Freezer
- Block I/O
- Devices

The CPU cgroup can use two different types of scheduler: either the **Completely Fair Scheduler (CFS)**, which is based on distributing CPU based on a weighting system. The **Real-Time Scheduler (RTS)** is the other alternative, and is a task scheduler that caps tasks based on their real-time utilization.

The memory cgroup is used to generate reports on memory utilization used by the tasks in a cgroup. It sets limits on the memory use of processes associated with the cgroup can use.

The freezer cgroup is used to control the process status of all processes associated with the freezer cgroup. The freezer cgroup can be used to control batches of jobs and issue the `FREEZE` command, which will stop all processes in the user space; the `THAW` command can be used to restart them again.

The **Block I/O (blkio)** cgroup monitors access to I/O on block devices and introduces limits on I/O bandwidth or access to resources. Blkio uses an I/O scheduler and can assign weights to distribute I/O or provide I/O throttling by setting maximum limits to throttle the amount of read or writes that a process can do on a device.

The devices' cgroup allows or denies access to devices by defining tasks under `devices.allow` and `devices.deny`, and can list device access using `devices.list`.

Benefits of containers

Containers have many benefits, with a focus on portability, agility, security, and as touched upon earlier in this chapter, have helped many organizations such as Netflix deploy their microservice architectures.

Containers also allow users to allocate different resources on an operating system using namespaces and limit CPU, memory, network block I/O, and network bandwidth using cgroups.

Containers are very quick to provision so can be scaled up and scaled down rapidly to allow elastic scaling in cloud environments. They can be scaled up rapidly to meet demand and containers can be migrated from one server to another using numerous techniques.

Cgroups can be configured quickly based on system changes, which gives users complete control over the low-level scheduling features of an operating system, which are normally delegated to the base operating system when using virtual machines or bare-metal servers. Containers can be tweaked to give greater fine-grained control over performance.

In some scenarios, not all resources on a bare-metal server will be utilized, which can be wasteful, so containers can be utilized to use all of the CPU and RAM available on a guest operating system by running multiple instances of the same application isolated by namespaces at a kernel level. This means that to each process, they appear to be functioning on their own unique operating system.

One of the main drawbacks with containers up until now has been that they have been notoriously low-level and hard to manage at scale. So, tooling such as for large implementation, and orchestration engines such as Docker Swarm, Google Kubernetes, and Apache Mesos alleviate that pain by creating abstraction layers to manage containers at scale.

Another benefit of containers is that they are very secure as they limit the attack surface area with additional layers of security added to the operating system through the use of different namespaces. If an operating system was compromised, an attacker would still need to compromise the system at the namespace level as opposed to having access to all processes.

Containers can be very useful when running multiple flavors of the same process; an example is a business that wants to run multiple versions of the same application for different customers. They want to prevent a spike in logins and transactions from one customer affecting another at the application level. Containers in this scenario would be a feasible solution.

Deploying containers

With the growing popularity of containers, traditional Linux distributions have been found to be sub-optimal and clunky when running a pure container platform.

As a result, very minimal operating systems have been created to host containers, such as CoreOS and Red Hat Atomic, which have been developed specifically to run containers.

Sharing information across operating systems is also a challenge for containers, as by design they are isolated by namespaces and cgroups to a particular host operating system. Key-value stores such as **etcd**, **Consul**, and **Zookeeper** can be used to cluster and cluster and share information across hosts.

CoreOS

CoreOS is a Linux-based operating system specifically created to provide a minimal operating system to run clusters of containers. It is the widest-used container operating system today and designed to run at massive scale without the need to frequently patch and update the software on the operating system manually.

Any application that runs on CoreOS will run in container format; CoreOS can run on bare-metal or virtual machines, on public and private clouds such as AWS and OpenStack.

CoreOS works by automatically pulling frequent security updates without affecting the containers running on the operating system. This means CoreOS doesn't need Linux admins to intervene and patch servers, as CoreOS automatically takes care of this by patching using its zero downtime security updates.

CoreOS focuses on moving application dependencies out of the application and into the container layer, so containers are dependent on other containers for their dependency management.

etcd

CoreOS uses etcd, which is a distributed key-value store that allows multiple containers across multiple machines to connect to it for data and state.

Etcd uses the **Raft algorithm** to elect a leader and uses followers to maintain consistency. When multiple etcd hosts are running, the state is pulled from the instance with the majority and propagated to the followers, so it is used to keep clusters consistent and up to date.

Applications can read and write data into etcd and it is designed to deal with fault and failure conditions. Etcd can be used to store connection strings to endpoints or other environment-specific data stores.

Docker

It would be impossible to talk about containers without mentioning Docker. In 2013, Docker was released as an open-source initiative that could be used to package and distribute containers. Docker was originally based on Linux LXC containers, but the Docker project has since drifted away from that standard as it has become more opinionated and mature.

Docker works on the principle of isolating a single process per container in the Linux kernel. Docker uses a union-capable file system, cgroups, and kernel namespaces to run containers and isolate processes. It has a command-line interface and a well thought out workflow.

Docker registry

When container images are packaged, they need to be pushed to Docker's container registry server, which is an image repository for containers.

The **Docker registry** is used to store containers, which can be tagged and versioned much like a package repository. This allows different container versions to be stored for roll-forward and roll-back purposes.

By default, the Docker registry is a file-system volume and persists data on a local file system. Artifact repositories such as Artifactory and Nexus now support Docker registry as a repository type. The Docker registry can be set up with authentication and SSL certificates to secure container images.

Docker daemon

During installation, Docker deploys a daemon on the target operating system that has been chosen to run containers. The **Docker daemon** is used to communicate with the Docker image registry and issue pull commands to pull down the latest container images or a specific tagged version. The Docker command line can then be used to schedule the start-up of the containers using the container image that has been pulled from the registry. Docker daemons, by default, run as a constant process on target operating systems, but can be started or stopped using a process manager such as `systemd`.

Packaging containers

Containers can be packaged in various different ways; two of the most popular ways of packaging containers is using Dockerfiles, and one of the lesser known ways is using a tool from **HashiCorp** called Packer. Both have slightly different approaches to packaging container images.

Dockerfile

Docker allows users to package containers using its very own configuration-management tool called **Dockerfile**. Dockerfile will state the intent of the container by outlining the packages that should be installed on it using package managers at build time.

The following Dockerfile shows NGINX being installed on CentOS by issuing `yum install` commands and exposing port 80 to the guest operating system from the packaged container. Port 80 is exposed so NGINX can be accessed externally:

```
RUN yum -y update; yum clean all
RUN yum -y install epel-release; yum clean all
RUN yum -y install nginx; yum clean all
RUN echo "daemon off;" >> /etc/nginx/nginx.conf
RUN echo "nginx on CentOS 6 inside Docker" > /usr/share/nginx/html/index.html

EXPOSE 80

CMD [ "/usr/sbin/nginx" ]
```

Once the Dockerfile has been created, Docker's command-line interface allows users to issue the following command to build a container:

```
docker build nginx
```

The one downside is that applications are typically installed using configuration management tools such as Puppet, Chef, Ansible, and Salt. The Dockerfile is very brittle, which means that packaging scripts need to be completely re-written.

Packer-Docker integration

Packer from HashiCorp is a command-line tool which uses multiple drivers to package virtual machine images and also supports creating Docker image files. Packer can be used to package **Amazon Machine Image (AMI)** images for AWS or **QEMU Copy On Write (QCOW)** images, which can be uploaded to OpenStack Glance.

When utilizing Packer, it skips the need for using Dockerfiles to create Docker images; instead, existing configuration-management tools such as Puppet, Chef, Ansible, and Salt can be used to provision and package Docker container images.

Packer has the following high-level architecture and uses a JSON file to describe the Packer workflow, with three main parts:

- Builders
- Provisioners
- Post-processors

Builders are used to boot an ISO, virtual machine on a Cloud platform, or in this case, start a Docker container from an image file on a build server.

Once booted, the configuration management **provisioner** will run a set of installation steps. This will create the desired state for the image, emulating what the Dockerfile would carry out. Once complete, the image will be stopped and packaged.

A set of **post-processors** will then be executed to push the image to an artifact repository or Docker registry, where it is tagged and versioned.

Using Packer means existing configuration management tools can be used to package virtual machines and containers in the same way rather than using a completely different configuration-management mechanism for containers. The Docker daemon will need to be installed as a prerequisite on the build server that is being used to package the container.

In the following example, an `nginx.json` Packer file is created; the `builders` section has the type `docker` defined, which lets Packer know to use the Docker builder.

The `export_path` is where the final Docker image will be exported to and `image` is the name of the Docker image file that will be pulled from the Docker registry and started.

One provisioner of the `ansible-local` type will then execute the `install_nginx.yml` playbook to install NGINX on the Docker image, using an Ansible playbook as opposed to the Dockerfile.

Finally, the post-processors will then import the packed image, complete with NGINX installed, into the Docker registry with the tag `1.1`:

```
{
  "builders": [{
    "type": "docker",
    "image": "centos6",
    "export_path": "image.tar"
  }],
  "provisioners": [
    {
      "type": "ansible-local",
      "playbook_file": "playbooks/install_nginx.yml"
    }
  ],
  "post-processors": [
    {
      "type": "docker-import",
      "repository": "image/releases",
      "tag": "1.1"
    }
  ]
}
```

To execute the Packer build, simply execute the following command passing the `nginx.json` file:

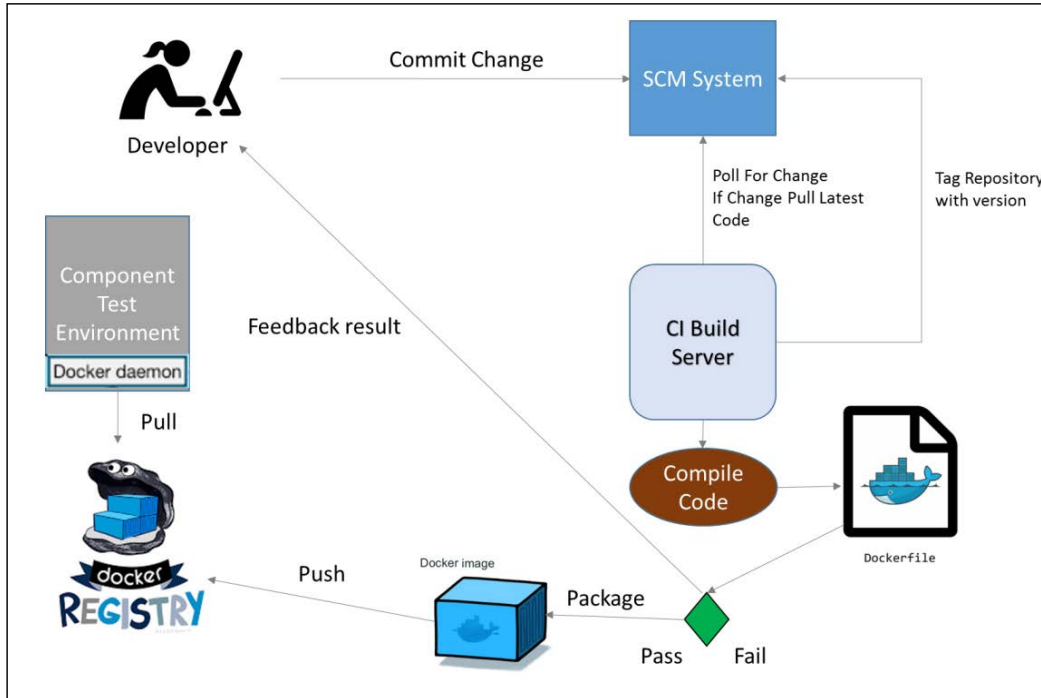
```
packer build nginx.json
```

Docker workflow

The **Docker workflow** fits nicely into the continuous integration process that we covered as part of *Chapter 7, Using Continuous Integration Builds for Network Configuration* and the Continuous Delivery workflow we covered in *Chapter 9, Using Continuous Delivery Pipelines to Deploy Network Changes*. After a developer pushes a new code commit, compiling and potentially packaging new code, the continuous integration process can be extended to execute a Dockerfile to package a new Docker image as a post-deployment step.

A Docker daemon is configured on each downstream test environment and production as part of the base operating system. At deployment time, the Docker daemon is scheduled to pull down the newly packaged Docker image and create a new set of containers doing a rolling update.

This process flow can be seen as follows:



Default Docker networking

In terms of networking, when Docker is installed, it creates three default networks; the networks created are the `bridge`, `none`, and `host` networks, as shown in the following screenshot:

```
$ docker network ls

NETWORK ID          NAME                DRIVER
7d456gs89ab6       bridge             bridge
3e202ee27b14       none               null
8f04fm033fb9       host               host
```

The Docker daemon creates containers against the `bridge` (`docker0`) network by default; this occurs when a `docker create` and `docker start` are issued on the target operating system, or alternatively, just a `docker run` command can be issued. These commands will create and start new containers on the host operating system from the defined Docker image.

The `none` network is used to create a container-specific network, which allows containers to be launched and left to run; it doesn't have a network interface, though. The `host` network adds containers to the same network as the guest operating system.

When containers are launched on it, Docker's bridge network assigns each container a unique IP address on the bridge network's subnet range. The containers can be viewed by issuing the following `docker network inspect` command:

```
docker network inspect bridge
```

Docker allows users to inspect container configuration by using the `docker attach` command; in this instance, the `nginx` container can be inspected:

```
docker attach nginx
```

Once attached, the `/etc/hosts` file can be inspected to show the network configuration. Docker bridge uses a NAT network and can use port forwarding using the following `-p` command-line argument. For example, `-p 8080:8080` forwards port 8080 from the host to the container. This allows all containers that are running on an operating system to be accessed directly by the localhost by their IPs, using port forwarding.

In its default networking mode, Docker allows containers to be interconnected using a `--links` command-line argument, which is used to connect containers, which writes entries into the `/etc/hosts` file of containers.

The default network setup is now not recommended for use, and more sophisticated networking is present, but the concepts it covers are still important.

Docker allows user-defined networks to be defined to host containers, using network drivers to create custom networks such as custom `bridge`, `overlay`, or `layer 2 MACVLAN` network.

Docker user-defined bridge network

A user-defined bridge network is much like the default Docker network, but it means that each container can talk to each of the other containers on the same bridge network; there is no need for linking as with the default Docker networking.

To place containers on a user-defined network, containers can be launched on the `devops_for_networking_bridge` user-defined bridge network using the following command, with the `-net` option set:

```
docker run -d -name load_balancer -net devops_for_networking_bridge nginx
```

Each container that is launched will reside on the same operating system guest. Publish is used to expose specific using of the `-p 8080-8081:8080/tcp` command. Therefore, ranges can be published so that portions of the network can be exposed.

Docker Swarm

Overlay networks, can also be used with Docker and have already been covered at length in this book, are a virtualized abstraction layer for the network. Docker can create an overlay network for containers, which is used to create a network of containers that belong to multiple different operating system hosts.

Instead of isolating each container to a unique network existing on one host, Docker instead allows its overlay network to join multiple different clusters of containers that are deployed on separate hosts together.

This means that each container that shares an overlay network will have a unique IP address and name. To create an overlay network, Docker uses its own orchestration engine, called **Docker Swarm**.

To run Docker in swarm mode, an external key-value store such as etcd, Consul, or Zookeeper needs to be used with Docker. This key-value store allows Docker to share information between different hosts, including the shared overlay network.

Docker machine

It is worth mentioning that `docker-machine` is a useful command-line utility that allows virtual machines to be provisioned in VirtualBox, OpenStack, AWS, and many more platforms that have drivers.

In the following example, we can see how a machine could be booted using `docker-machine` in OpenStack:

```
docker-machine create -driver openstack (boot arguments and credentials)
docker-dev
```

One of the more useful functions of `docker-machine` is its ability to boot virtual machines in cloud environments while issuing Docker Swarm commands. This allows machines to be set up on boot to the specific profile that is required.

Docker Compose

Another helpful tool for orchestrating containers is **Docker Compose**, as running a command line for every container that needs to be deployed is not a feasible solution at scale. Therefore, Docker Compose allows users to specify their microservice-architecture topology in YAML format, so container dependencies are chained together to form a fully-fledged application.

Microservices will be comprised of different container types, which together make up a full application. Docker Compose allows each of those microservices to be defined as YAML in the `docker-compose` file so they can be deployed together in a manageable way.

In the following `docker-compose.yml` file, `web`, `nginx`, and `db` applications are configured and linked together, with the load balancer being exposed on port 8080 for public access, and load balancing `app1`, which is connected to the `redis` database backend:

```
web:
  build: ./app1
  volumes:
    - "./app:/src/app1"
  ports:
    - "8080:8080"
  links:
    - "db:redis"
  command: init -L app1/bin

nginx:
  build: ./nginx/
  ports:
    - "800:80"
  volumes:
    - /www/public
  volumes_from:
    - web
  links:
    - web:web

db:
  image: redis
```

Docker Compose can be executed in the same directory as the Docker Compose YAML file to invoke a new deployment the following command should be issued:

```
docker-compose up
```

Swarm architecture

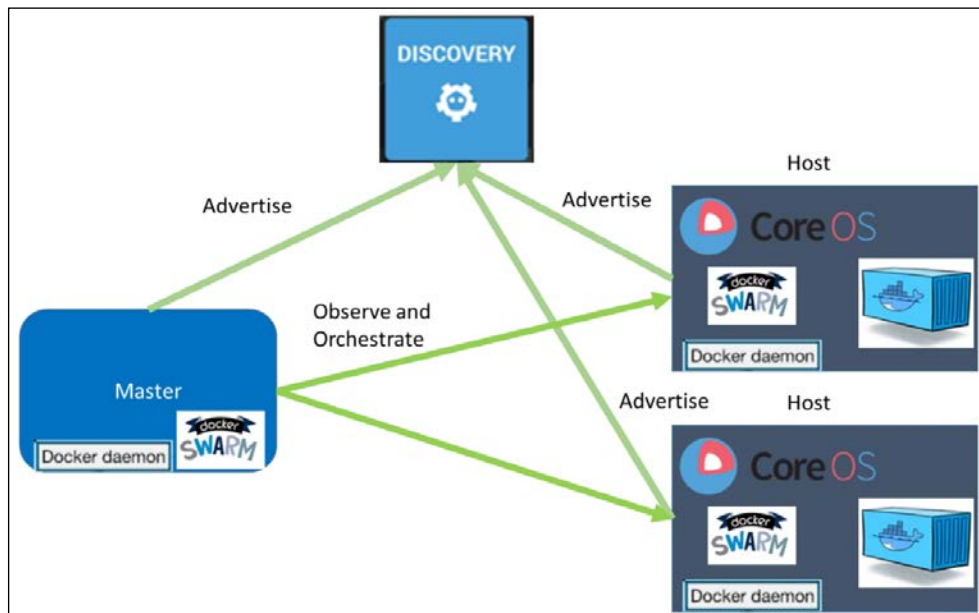
The Swarm architecture works on the principle that each host runs a Swarm agent and one host runs a Swarm master. The master is responsible for the orchestration of containers on each of the hosts where agents are running and that are a member of the same discovery (key-value store).

An important principle for swarm is discovery, which is catered for using a key-value store such as etcd, Consul, or Zookeeper.

To set up a Docker swarm, a set up Docker machine can be used to provision the following:

- Discovery server (key-value store such as etcd, Consul, or Zookeeper)
- Swarm master with swarm agent installed, pointing at a key-value store
- Two Swarm nodes with Swarm agent installed, pointing at a key-value store

The Docker Swarm architecture shows a master node scheduling containers on two Docker agents while they are all advertising to the key-value store, which is used for service discovery:



When setting up a Swarm agent, in this case the Swarm master, they will be booted with the following options: `--swarm-discovery` defines the address of the discovery service, while `--cluster-advertise` advertises the host machine on the network and `--cluster-store` points at a key-value store of choice:

```
docker-machine create -d openstack (boot arguments and credentials) --swarm
--swarm-master --swarm-discovery="consul://10.100.100.10:8500"
--engine-opt="cluster-store=consul://10.100.100.10:8500"
--engine-opt="cluster-advertise=eth1:2376"
swarm-master
```

Once the architecture has been set up, an overlay network needs to be created to run containers across the two different hosts (in this instance the overlay network is called `devops_for_networking_overlay`) by issuing the following command:

```
docker network create -d overlay devops_for_networking_overlay
```

Containers can then be created on the network from an image using the Docker Swarm master to schedule the commands:

```
docker run -d --name loadbalancer --net devops_for_networking_overlay nginx
```

As each host is running in Swarm mode and attached to the key-value store, upon creation, the network information meta-data will be shared by the key-value store. This means that the network is visible to all hosts that use the same key-value store.

Containers can then be launched from any of the Swarm masters onto the same overlay network, which will join the two hosts together. This will allow each host to communicate with other containers, via the overlay network, across hosts.

Multiple overlay networks can be created; though containers can only communicate across the same overlay network they cannot communicate between different overlay networks. To mitigate this, containers can be attached to multiple different networks.

Docker Swarm allows many specific containers to be assigned and exposed using port forwarding to load balance containers. Rolling updates can also be carried out to allow upgrades of the containers' application version.

Due to its completely decentralized design, Docker Swarm is very flexible in the number of networking use cases it can solve.

Kubernetes

Kubernetes is a popular container orchestration tool from Google which was created in 2014 and is an open-source tool. Rather than Google coming up with their own container packaging tool and packaging repository, Kubernetes instead can plug seamlessly to use Docker registry as its container image repository.

Kubernetes can orchestrate containers that are created using Docker via a **Dockerfile**, or alternatively, using Packer aided by configuration management tools such as Puppet, Chef, Ansible, and Salt.

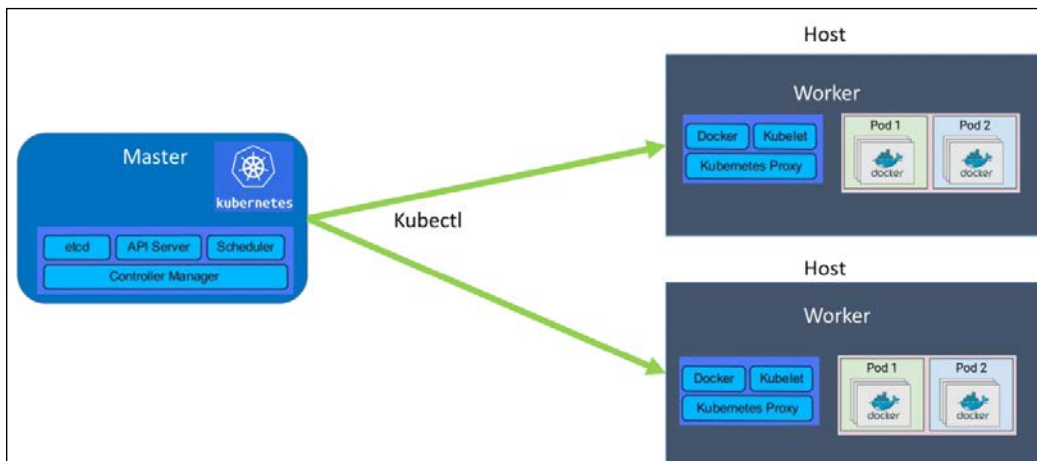
Kubernetes can be seen as an alternative to Docker Swarm, but takes a slightly different approach in terms of its architectural design and has a lot of rich scheduling features to help with container management.

Kubernetes architecture

A Kubernetes cluster needs to be set up before a user can use Kubernetes to schedule containers. There is a wide variety of configuration management tools that can be used to create a production-grade Kubernetes cluster with notable solutions available from Ansible, Chef, and Puppet.

Kubernetes clustering consists of the following high-level components, which in turn have their own subset of services. At a high level, a Kubernetes cluster consists of the following components:

- Kubectl
- Master node
- Worker node



Kubernetes master node

The master node is responsible for managing the whole Kubernetes cluster and is used to take care of orchestrating worker nodes, which is where containers are scheduled.

The master node, when deployed, consists of the following high-level components:

- API server
- Etcd key-value store
- Scheduler
- Controller manager

The API server has a RESTful API, which allows administrators to issue commands to Kubernetes.

Etcd, as covered earlier in this chapter, is a key-value store that allows Kubernetes to store state and push changes to the rest of the cluster after changes have been made. Etcd is used by Kubernetes to hold scheduling information about pods, services, state, or even namespace information.

The Kubernetes scheduler, as the name suggests, is used to schedule containers on Services or Pods. The Scheduler will check the availability of the Kubernetes cluster and make scheduling decisions based on availability of resources so it can schedule containers appropriately.

The controller-manager is a daemon that allows a Kubernetes master to run different controller types. Controllers are used by Kubernetes to analyze the state of a cluster and make sure it is in the desired state, so if a pod fails it will be recreated or re-started. It adheres to the thresholds that are specified and is controlled by the Kubernetes' administrator.

Kubernetes worker node

Worker nodes are where pods run; each pod has an IP address and runs containers. It is the pod that determines all the networking for the containers and governs how they communicate across different pods.

The worker node will contain all the necessary services to manage the networking between the containers, communicate with the master node, and are also used to assign resources to the scheduled containers.

Docker also runs on each of the worker nodes and is used to pull down containers from the Docker registry and schedule containers.

Kubelet is the worker service and is installed on worker nodes. It communicates with the API server on the Kubernetes master and retrieves information on the desired state of pods. Kubelet also reads information updates from etcd and writes updates about cluster events.

The `kube-proxy` takes care of load balancing and networking functions such as routing packets.

Kubernetes kubectl

Kubectl is the Kubernetes command line, which issues commands to the master node to administer Kubernetes clusters. It can also be used to call YAML or JSON, as it is talking to the RESTful API server on the master node.

A Kubernetes service is created as an abstraction layer above pods, which can be targeted using a label selector.

In the following example, `kubectl` can be used to create a `loadbalancing_service` service deployment with a selector, `app: nginx`, which is defined by the `loadbalancing_service.yml` file:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: loadbalancing_service
    name: loadbalancing_service
spec:
  ports:
    - port: 81
service.
  selector:
    app: nginx
  type: LoadBalancer
```

Kubectl executes the YAML file by specifying:

```
kubectl create -f loadbalancing_service.yml
```

Kubectl can then create four replica pods using the `ReplicationController`, these four pods will be managed by the service, as the labels `app: nginx` match the service's selector and launch an NGINX container in each pod using the `nginx_pod.yml` file:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 4
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx_custom
        ports:
        - containerPort: 80
```

Kubect1 creates the service using the following:

```
kubect1 create -f nginx_pod.yml
```

Kubernetes SDN integration

Kubernetes supports multiple networking techniques that could fill a whole book's worth of material on its own. With the Kubernetes, the pod is the major insertion point for networking.

Kubernetes supports the following networking options:

- Google Compute Engine
- Open vSwitch
- Layer 2 Linux Bridge
- Project Calico
- Romana
- Contiv

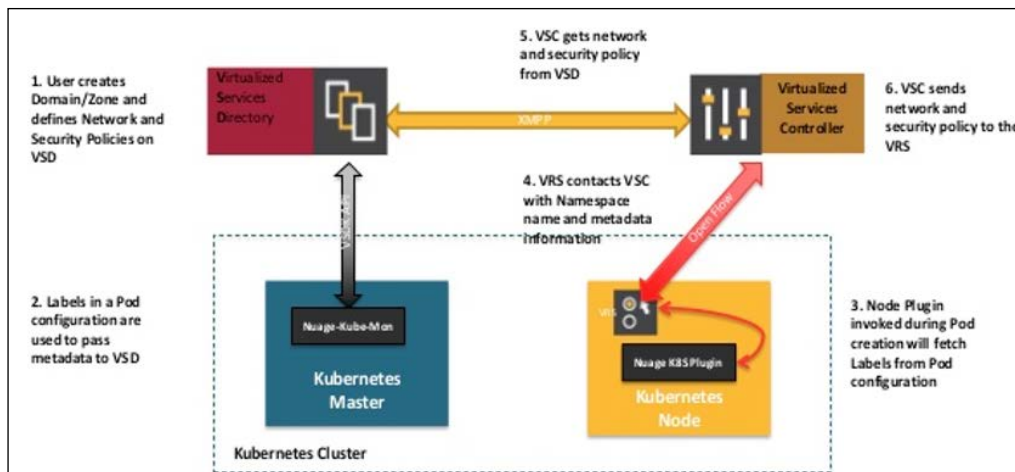
Kubernetes looks to provide a pluggable framework to control a pod's networking configuration and aims to give users a choice; if a flat layer 2 is required, Kubernetes caters for it, if a more complex layer-3 overlay network is required, then it can cater for this, too.

With Open vSwitch being widely used with enterprise SDN controllers such as Nuage Networks VSP platform, which was covered in *Chapter 2, The Emergence of Software-defined Networking* and *Chapter 6, Orchestrating SDN Controllers Using Ansible*. This focused upon how flow information could be pushed down to Open vSwitch on each hypervisor to create a stateful firewall and govern the ACL policies.

A similar implementation is carried out when integrating Kubernetes, with Open vSwitch, being deployed onto each worker node and pod traffic being deferred to Open vSwitch.

In Nuage's case, a version of their customized version of Open vSwitch, known as the VRS, is deployed on each Kubernetes worker to govern policy controlled by the VSD Nuage VSPs policy engine..

The workflow for the Nuage SDN integration with Kubernetes is shown in the following figure, which shows that enterprise SDN controllers can integrate with orchestration engines such as Kubernetes and Docker to provide enterprise-grade networking:



Impact of containers on networking

Containers have undoubtedly meant that a lot of networking has shifted into the application tier, so really, containers can be seen as a PaaS offering in its truest form.

Infrastructure is, of course, still required to run containers, be it on bare-metal servers or virtual machines. The merits of virtual machines being used to run containers long term are debatable, as in a way it means a double set of virtualization, and anyone using nested virtualization will know it isn't always optimal for performance. So with more organizations using containers to deploy their microservice architectures, it will undoubtedly mean that users having a choice to run containers on either virtual or physical machines will be in demand.

Cloud has notoriously meant virtual machines, so running containers on virtual machines is probably born out of necessity rather than choice. Being able to orchestrate containers on bare-metal servers with an overlay network on top of them is definitely more appealing as it pushes the container closer to the physical machine resources without the virtualization overhead.

This allows containers to maximize the physical machine resources, and users then only care about anti-infinity in terms of whether the service can run across multiple clouds and data centers, giving true disaster recovery.

With hybrid cloud solutions, the industry is moving beyond thinking about rack redundancy. Instead it is moving toward a model which will focus on splitting applications across multiple cloud providers. So having the ability to orchestrate the networking and applications in an identical way using orchestration engines such as Docker Swarm or Kubernetes can be used to make that goal a reality.

What does this mean for the network operator? It means that the role is evolving, it means that the network engineer's role becomes advisory, helping the developers architect the network in the best possible way to run their applications. Rather than building a network as a side project in a private cloud, network operators can instead focus on providing an overlay network as a service to developers while making the underlay network fabric fast and performant so that it can scale out to meet the developer's needs.

Summary

Containers have been said to be a major disruptor of the virtualization market. Gartner have predicted the following:

"By 2018, more than 50-60% of new workloads will be deployed into containers in at least one stage of the application life cycle".

This is based on Gartner's analysis of the IT market, so this is a bold statement, but if it comes to fruition, it will prove to be a huge cultural shift in the way applications are deployed, in the same way virtualization was before it.

In this chapter, we showed that containers can help organizations deploy their microservice architectures and analyzed the internal mechanics and benefits that containers bring. The key benefits are portability, speed of deployment, elastic scalability, isolation and maximization of different resources, performance control, limited attack vector, and support for multiple networking types.

Aside from the benefits containers bring, this chapter looked at the Docker tool and illustrated how the Docker workflow can be fitted into a Continuous Delivery model, which is at the heart of most DevOps initiatives.

The focus of the chapter then shifted to Docker networking and the layer-2 networking options available to network containers. We illustrated how to use overlay networks to join multiple hosts together to form a cluster and we showed how container technology can integrate with SDN controllers such as Nuage VSP Platform using Open vSwitch.

The chapter also covered container orchestration solutions such as Docker Swarm and Kubernetes, their unique architectures, and ways in which they can be used to network containers over multiple hosts and act as a Platform as a Service layer.

The importance of containerization and its impact on Platform as a Service (PaaS) solutions cannot be underestimated, with Forrester stating the following:

"Containers as a Service (CaaS) is becoming the new Platform as a Service (PaaS). With the interest in containers and micro-services skyrocketing among developers, cloud providers are capitalizing on the opportunity through hosted container management services."

In summary, it is fair to conclude that containerization can have many benefits and help aid developers in the implementation of Continuous Delivery workflows and PaaS solutions. Containerization also gives the added flexibility of deploying workloads across multiple cloud providers, be they private or public, using a common orchestration layer such as Kubernetes, Apache Mesos, or Docker Swarm.

In the following chapter, the focus will shift from containers toward securing the network when using software-defined overlay networks and a Continuous Delivery model. It will explore techniques that can be used to help secure a modern private cloud in an API-driven environment, so that software-defined networking solutions can be implemented without compromising security requirements.

11

Securing the Network

With many businesses transitioning to software-defined networks and using APIs to make network changes, the importance of securing the network is a prominent concern. Security implementations need to evolve too, as the network is virtualized and modern protocols are used to build Leaf-Spine architectures to scale out multi-tenant cloud environments.

In this chapter, the following topics will be covered:

- The evolution of network security and debunking myths
- Securing a software-defined network
- Network security and Continuous Delivery

The evolution of network security and debunking myths

As network engineers become accustomed to a flat layer 2 network and Spanning Tree protocol as discussed in *Chapter 1, The Impact of Cloud on Networking*, network security and approaches towards securing an enterprise network have become very mature and well understood by security teams over the years.

Most security engineers are well versed in the best practices that should be implemented when dealing with physical networks. A security team will normally look to implement a rigid set of security best practices on the network, which network teams must comply with, to pass necessary accreditations. But how applicable are these best practices when implementing software-defined networking?

It is fair to say that there is still a knowledge gap that exists regarding software-defined networking at the moment and there is a degree of fear and uncertainty of the unknown from security engineers and even some network engineers.

This chapter will hopefully help demystify some of those concerns. This is coming from someone that helps run a software-defined network in production, so this isn't talking about theories or aspirations, it is based on hard facts.

So first let's review some of the requests from security teams around network security and look at how these requests should be adapted when dealing with software-defined networking.

Account management

In terms of account management, security teams will normally stipulate that the following best practices are adhered to when setting up user access:

- Two-factor authentication should be used when accessing production servers
- User accounts should respect the least privileges necessary for users
- Unique user accounts should be used between test and production environments
- **DenyAll** should be default on Access Control Lists
- Use **Terminal Access Controller Access Control System (TACACS)** or equivalent authentication to access network devices

When reviewing account management, all points outlined by security practitioners remain valid. Any **Software-defined Networking (SDN)** controller or modern switch vendor should meet the account management requirements when they are being evaluated. If they don't meet these requirements, they simply should not be implemented in production.

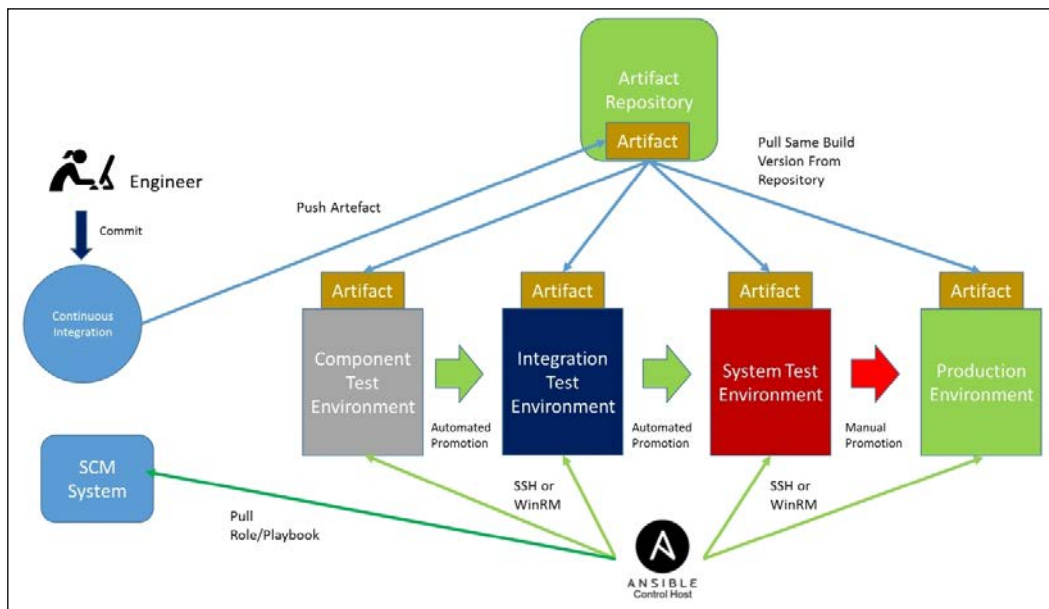
When using software-defined networking always aligned with a Continuous Delivery model, service accounts will be used by orchestration and configuration management tools such as **Ansible** to setup subnets, networks, and **ACL** policies to carry out any network changes.

The user committed to the source control management system will inevitably be the person that invoked the network changes even though it will be invoked by a service account. This in itself is a cultural shift, so privileges on the source control management repository should be reviewed to set up **Active Directory Domain Services (ADDS)** or **Lightweight Directory Access Protocol (LDAP)** access so commits are tracked and can be traced back to the user that made the change. Continuous Delivery tooling such as **Jenkins**, **ThoughtWorks, Go**, or a Plethora of other continuous integration build servers can cater for this requirement.

Separate service accounts can be used to orchestrate test and production environments to meet security best practices. All other user accounts in a Continuous Delivery model should be read-only, so users can view the outcome of a Continuous Delivery deployment, which is driven by automation, with a break glass account being the exception to the rule and available for use in a state of emergency.

It is important for security teams to understand that if users are manually intervening in immutable software-defined networks, then the overall Continuous Delivery model could break, so there is no appetite to do manual configuration. All desired state should be controlled via source control management systems and pushed out to systems accordingly. This, again, is a mindset change and security practitioners generally find this hard to believe, as they have spent years seeing network engineers push manual changes to devices to make any changes, but this is a new approach and a huge change for some.

This was illustrated in *Chapter 9, Using Continuous Delivery Pipelines to Deploy Network Changes*, when utilizing configuration management tools such as Ansible to push out network changes to test and production environments:



This concept is initially difficult for security practitioners to grasp, but the **DevSecOps** movement is helping security practitioners see the window of opportunity that an automated Continuous Delivery process brings.

Automation should mean users have fewer individual privileges and approved workflow actions are hardened and signed off, which govern what kind of interaction is allowed. All of this is controlled via the Continuous Delivery pipeline.

Network device configuration

Security best practices with regards to the configuration of network devices focus on keeping an up-to-date auditable inventory of the network, with security patching being applied on a regular basis to each network device at an operating system level, and secure protocols and **Public Key Infrastructure (PKI)** certificates being applied from relevant trust stores.

As such, a common set of requirements from a security team for configuration of network devices may include:

- A network hardware list should be available in an **IP Address Management (IPAM)** solution
- All network devices should be patched regularly
- Configure SNMP version 3 or above
- Disable ports not in use
- Use **Transport Layer Security (TLS)** to encrypt network traffic

The security approaches to network device configuration should also not change in terms of setup, as SDN controllers and modern switch vendors should look to use TLS, be patched regularly, and be accessible via DNS.

An underlay network when utilizing a Leaf-Spine architecture and overlay network is still comprised of physical network devices, so the configuration and best practices associated with securing these devices are still completely valid and integral.

SDN controllers, like network switches, are deployed on the layer 2 underlay network, so should follow the same conventions as network switches, have secure protocols, and adhere to patching schedules.

Firewalling

One of the major sources of confusion from security teams in industry when looking at software-defined networking seems to be around firewalling and some fear and uncertainty exists as they are used to using physical **stateful** firewalls in production networks.

However, as long as a virtual firewall meets security requirements, there should be no issue implementing SDN controllers and allowing them to control firewalling and segmentation of the network using virtualized micro-segmentation policies.

Security teams will traditionally mandate the following requirements from firewalls:

- Use a stateful firewall
- Use explicit permits and implicit denies on ACL rules
- Have the ability to audit teams' ACL access
- Log all denied attempts on the firewall

Firewall best practices should always be adhered to when implementing software-defined networking; traditionally though, security teams have always pushed for stateful physical firewalls to separate three-tier models, which are segregated into frontend, business logic, and backend tiers.

With the move towards microservices and the adoption of software-defined networking, applications have tended not to fit into this structure and **Open vSwitch** has allowed OpenFlow to be used to implement **Ingress** and **Egress** policies at the hypervisor level or operating system host level.

We have also seen that this same process can be applied to containers in *Chapter 11, The Impact of Containers on Networking*, and Open VSwitch can be installed on container hosts such as Core OS, or even on bare metal servers to control firewall policies.

As long as the same best practice principles of using explicit permits and implicit denies on ACL rules are adhered to, and a process is set up to log all denied attempts on the firewall, then there should be no reasons to argue against the merits of using virtualized firewalling.

Open vSwitch now offers stateful firewalling, which is now as secure as iptables on a Linux operating system or a physical firewall, so there is now no reason why firewalling cannot be virtualized for enterprise networks. This mirrors the debate about the use of hypervisors initially for infrastructure services, but the gains it brings a business in terms of scalability, programmability, auditability, and manageability make it hard to argue against firewall virtualization.

Vulnerability detection

Overlay networks, in terms of the detection of vulnerabilities and attacks, should also not change in terms of security requirements, although the method for acquiring the data may need to change slightly, as protocols such as **Border Gateway Protocol (BGP)** and **Virtual Extensible LAN (VXLAN)** need different tooling to track packets in the network.

When looking at vulnerability detection and data sampling, the following activities should be scheduled on a regular basis:

- Regular vulnerability scanning
- Deep packet inspection

In terms of vulnerability scanning, scanning of the network and network devices should be carried out frequently. Ideally security scanners themselves should have separate responsibilities in a software-defined network. A security scanner should have access to the underlay to do a full scan and another profile of the scanner should be used for the overlay. If a scanner has access to the underlay and overlay, it becomes an attack vector, which if compromised would allow an attacker complete access to the network. So this is an important point often ignored; the overlay and underlay network devices and compute should not be routable to one another if possible.

It is often a requirement of a security team to be able to inspect network packets using deep packet inspection to make sure that there is no malicious activity. This has been done on flat layer 2 networks by inspecting packets that are transmitted between VLANs.

However, with overlay transporting packets using VXLAN encapsulation, networks can scale out network and alleviate the 4096 VLAN limit. This means that network and security teams will require tools that can de-encapsulate VXLAN packets so they can be able to inspect packets, as they have with VLAN packets, otherwise security tools will see data is being transmitted but it won't be able to be read it.

Setting up tooling that does VXLAN de-encapsulation is by no means an insurmountable challenge. Tooling is available to do this, it will just require that network and security teams alter the tools they are currently used to using.

Network segmentation

One of the biggest changes when implementing a software-defined overlay network is a shift away from the principles of a flat layer 2 network and VLAN segregation between networks.

Security teams are used to dealing with physical networks, so they will normally stipulate that the following requirements need to be met:

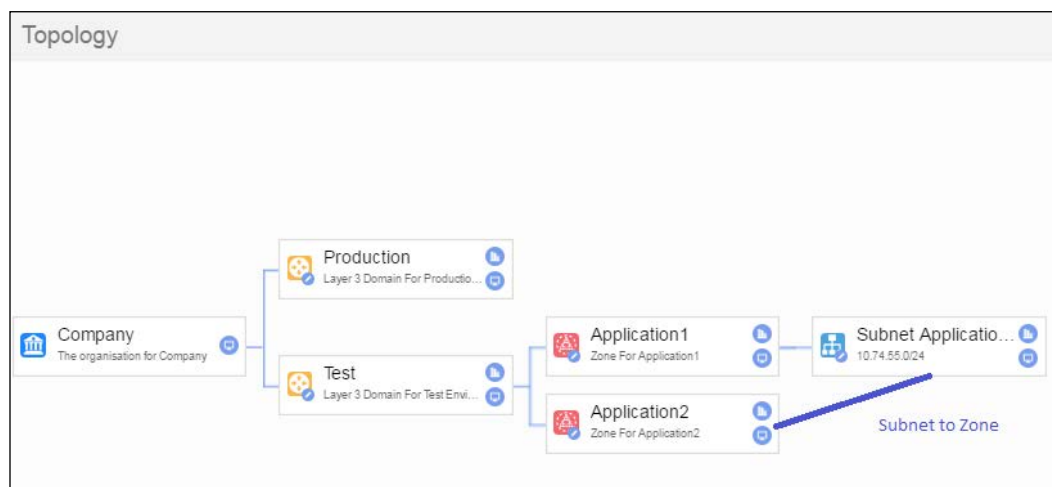
- Use VLANs to segregate traffic types (frontend, business logic, and backend)
- Ability to segregate **Test** and **Production**
- Use firewalls between different network tiers

However, SDN controllers create VXLAN tunnels between hardware **Virtual Tunnel End Points (VTEPs)** on network switches and stretches them to each hardware compute node to build a virtualized overlay underlay network over the network.

SDN controllers are used to translate the **Open vSwitch Database (OVSDB)** information from switch vendors and push the flow data down to each compute node (hypervisor, container, or bare metal server), which is dictated by the SDN controller's policy engine to create firewalls and micro-segmentation.

It is a differing approach; firewalling per microservice application is dictated by **OpenFlow** and used to control Ingress and Egress policies. Using overlay networks, applications can communicate with another application's micro-segmented zone as illustrated by the **Nuage Networks Virtual Service Platform (VSP)** in *Chapter 2, The Emergence Of Software, Defined Networking*.

In the following example, we can see the **Application 1** micro-subnet communicating with **Application 2** by communicating subnet to zone:



Using micro-segmentation of firewall rules per application moves away from having physical, stateful firewalls segmenting zones for all applications. Instead, individual firewalls are created per application to govern segmentation between the network, with layer 3 domains segmenting **Test** and **Production** from each other at a layer above.

Each application has their own policy in this micro-segmentation model, which means security teams have the ability to audit firewall policies and understand what each application is communicating with.

Overlay networks for this reason should bring security gains, as it becomes completely clear the connectivity requirements for an application and connectivity topologies are not lost in a set of monolithic ACL rules on a physical stateful firewall, which aren't clearly mapped to each application.

Software-defined networking should mean that each application has an initial deny all and opens up only the minimum amount of explicit access so they can access other applications or services in the network.

This is far more secure than opening up port ranges on a stateful firewall, so overlay networks should, in theory, improve complex network security when implemented correctly. If immutable networks are used as highlighted by A/B subnets in *Chapter 6, Orchestrating SDN Controllers Using Ansible*, then automatic cleanup of old ACL rules is also implemented by default, which has been a challenge for network and security teams as they are afraid to remove old policies in the fear of creating an outage for a particular application.

Security teams can audit that policy with development teams and advise on any changes that need to be made, safe in the knowledge that as far as a development team is concerned, all of the ACL policies they are implementing are required to deploy their application with the bare minimum amount of explicit Ingress and Egress ACL rules being used.

Securing a software-defined network

So far in this chapter, we have focused on a set of minimal network security requirements to make sure that a software-defined network is secure.

But to maximize the security of a software-defined network, we should look at how overlay and underlay networks could potentially be exploited in new ways by attackers and look at different mechanisms that can be put in place to prevent this from happening.

Software-defined Networks are split into the overlay (which holds all the virtualized networks that houses virtual, physical machines, and containers) and the underlay (which holds all bare metal machines such as hypervisors, network devices, and SDN controllers).

Attacks at Overlay

Overlay networks are created to allow networks to be automated programmatically via APIs and increase the speed of change by simplifying the network in software.

Within the remit of Continuous Delivery, self-service ACL rules can be set up by developers to govern north to south and east to west ACL policies.

It is important to have implicit controls that make sure that common workflow actions only allow teams to set ACL rules from their micro-subnet to different locations in the network, and that they can't compromise the integrity of any other network in the overlay except their own. So this should be demonstrable by testing the self-service automation to security teams.

Micro-segmentation is powerful in the following example:

When using implicit allows, team A with application 1 can only communicate with application 2, which is maintained by team B, if team B allows explicit Ingress rules that allow application 1 to communicate with it. So teams will have to coordinate between themselves, and their applications will only be able to communicate with one another if there is both an Egress and Ingress rule on each microservice application's firewall.

Aside from this, some applications may need northbound Internet access, so it is important that network teams put in place a mechanism to proxy out to the Internet and not give teams the ability to directly access it. A controlled proxy mechanism should be implemented by the network team so that there is a fixed mechanism to govern northbound Internet access.

Attackers may try and compromise a virtual machine or physical server that is part of the overlay network. Once they gain access to a machine, they could attempt to download software and compromise the network. An attacker could potentially attempt a **Denial of Service (DoS)** on a particular micro-subnet, which could be used to compromise a key service by compromising all virtual machines in the micro-subnet.

A benefit of micro-segmentation over a layer 2 network is that if one box was compromised in production, an attacker could have access to the whole frontend, business logic, or backend zone, while with micro-segmentation they would be isolated to the particular application.

With regards to outbound Internet access and setting up a proxy, it is imperative that upstream repositories used to download software packages to hosts go via a controlled proxy server, using an artifact repository with **Role Based Access Control (RBAC)** using Active Directory Domain Services or LDAP such as **Artifactory** or **Nexus**.

This means that servers within the overlay network can only access a set of approved third-party software repositories that have been given the blessing of the infrastructure team. Repositories not on the approved list cannot be accessed via the overlay network servers, as they are not proxied by the artifact repository, thus preventing the installation of dubious packages onto servers in the overlay network.

Proxying via an artifact repository means network and security teams can take measures to prevent packet sniffing software being downloaded onto a server to discover adjacent services or open ports dictated by the Ingress and Egress flow data.

It may also be desirable to disable **Internet Control Message Protocol (ICMP)** in the overlay network so that an attacker cannot work out the IP addresses of adjacent servers in a micro-subnet or underlay network devices such as top-of-rack switches and SDN controllers by doing a trace route.

If a server in the overlay network is logging drops, then appropriate alerting should be set up to notify the network or security team that some illicit activity is occurring within a micro-subnet.

Mechanisms can be put in place to tag compromised boxes with metadata in this case and use tools such as Ansible dynamic inventory to target them altogether by issuing a shut down or moving them to a quarantined network using live migration, which will stop a potential network attacker from gaining access to other servers in the network.

Attacks on the underlay network?

The **underlay** network could be targeted by potential attackers by gaining access to a hypervisor and looking to compromise Open vSwitch. This would allow them to directly instantiate new flows into the Open vSwitches flow-table, allowing access to multiple different locations in the network.

The attacker could sniff traffic and perform a **Man in the Middle (MitM)** attack on different network components as a result, so hypervisors should ideally be on a separate network, which will isolate access to compute servers and not allow them to be directly routable from the overlay network.

In the underlay network, switches now utilize centralized management systems to push updates to switches. For example, the **Arista CloudVision** platform **CloudVision eXchange (CVX)** servers are used to push configuration to all Arista switches, so it is imperative that access control to its API endpoints is done over HTTPs and that the management of switches is done on a completely dedicated network.

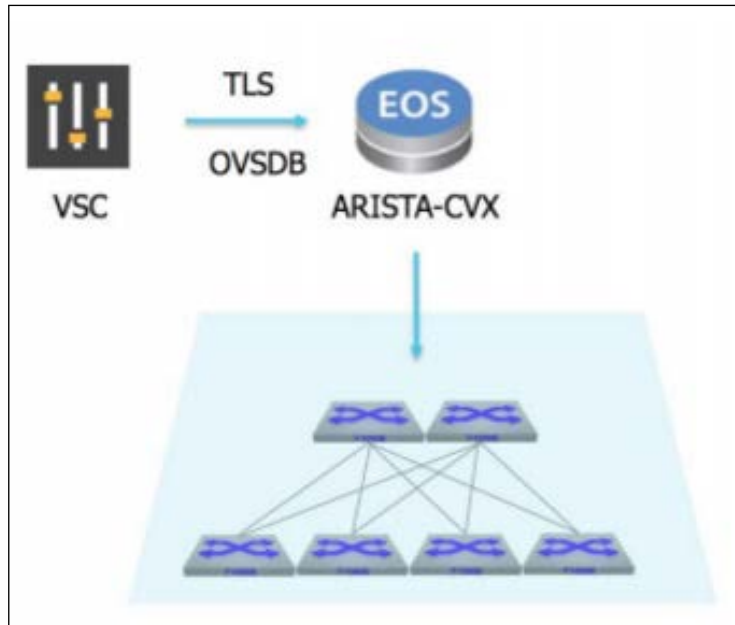
An attacker could potentially drop the whole configuration of every switch if the CVX cluster was compromised to create a DoS attack on the network, which would also mean that all routing would be dropped by the SDN controller.

An **Out Of Band (OOB)** network should ideally be implemented to govern access to network appliances with access provided via TACCs accounts. Using an OOB network for the northbound and southbound communications can help secure network devices and provide an extra degree of security for network devices.

The underlay and overlay network should be on completely different networks and not routable; this means that if a hypervisor is compromised in the underlay network, then an attacker will not be able to directly jump from an underlay box to the overlay. Underlay boxes should ideally be protected using bastion servers with two-factor authentication so no servers are directly accessible.

SDN controllers are typically x86 compute, and talk via REST API calls, so it should be mandatory to implement TLS on the SDN controllers and if possible issue a PKI CA to manage trust, authenticity, and revocation of access.

In the following example, we can see that the **Arista CVX** platform communicates with the Nuage **VSC** SDN controller using **OVSDB** with **TLS** on the underlay:



If underlay devices communicate using HTTP sessions, it will make the network susceptible to attacks in the Overlay network not just the underlay network.

Taking the **OpenStack** platform as an example, an SDN controller communicating with the OpenStack Neutron plug in will exchange all Ingress and Egress information for the entire overlay network. If this connection is using unencrypted REST API calls, it would mean that an attacker could intercept or track all flow information, and this can be used to compromise any number of tenant networks within the overlay.

Attacks on the SDN controller

The northbound API on an SDN controller is a desirable attack vector that could be used to compromise the whole overlay network.

To prevent this, RBAC should be put in place with sufficient password best practices adhered to. If the SDN controller's northbound API is compromised, then attackers could create new flow data programmatically against the overlay.

This would allow an attacker to traverse the network and target multiple services, allowing the attacker to bypass denying firewall policies and access multiple tenant networks.

Default admin accounts should have their passwords changed from day one, to avoid attackers guessing default accounts passwords. Complex passwords should be used at all times.

Audit trails should be set up on the SDN controller and logged to a **syslog** server, which will allow network and security engineers to check for unauthorized changes by attackers. If any irregular behavior occurs, then subsequent alerts should be triggered and the account should be disabled immediately.

On SDN controllers, SNMPv3 should be enabled as opposed to earlier versions and LDAP accounts, or SSH keys set up to allow access to Linux-based operating systems as opposed to using single service accounts or root access for underlay changes.

Network security and Continuous Delivery

Network security should be improved when using automation to push network changes out to network devices, or to change the desired state of overlay networks. It should increase the visibility of changes, as all changes are done from a centralized process, with no exceptions.

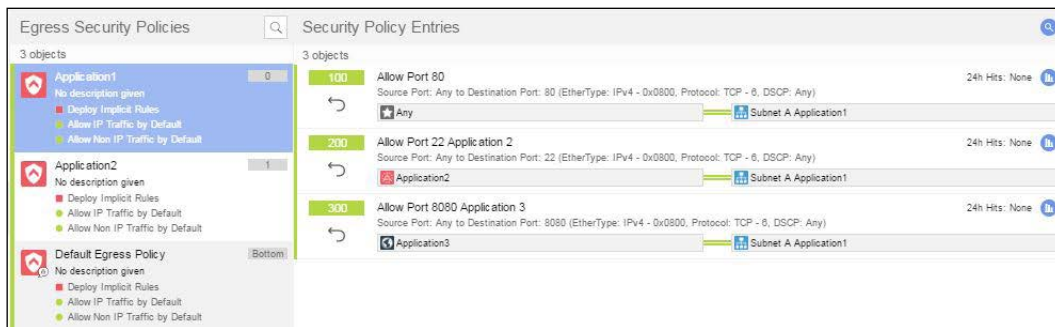
Continuous delivery processes, by design, should allow security teams to see clearly which user committed a network change. When a change is pushed to network devices or SDN controllers using the Continuous Delivery process, it will allow easy roll back to a previous version if the security team don't approve of the changes. However, this is still very reactive and continuous integration and delivery processes should include compliance and security checks as part of the continuous integration and delivery process.

Having compliance checks as part of Continuous Delivery provides a lot of flexibility for network and security teams. This will enable security teams to utilize some of the continuous integration and delivery best practices to help secure a network, such as continual testing and validation of changes integrated as part of the deployment pipeline.

Application connectivity topology

In a software-defined network, each application is micro-segmented, so they have individual application policies that can be audited by security or network teams. This will help with security compliance, as it allows security practitioners to see all the Ingress or Egress rules for a particular application in the overlay network.

This was highlighted in *Chapter 2, The Emergence Of Software-defined Networking*, showing micro-segmented policies per application with egress policies for **Application1** defined as shown:



The applications Ingress and Egress ACL rules should be readable and auditable in source control management systems using YAML files, or any other chosen configuration file used to control the SDN controller's desired state.

The live state of the system will also be present on SDN controller GUIs, which can be observed to make sure it matches what is defined in source control management systems.

It is important for security practitioners to be able to read and understand the configuration files that are being used to determine the current connectivity and state of the network.

Network and security teams have unique goals such as passing security audits to keep the business operational. It is important for Security teams to be able to see the application connectivity matrix and be able to have full visibility over connectivity.

For instance, when processing credit card transactions, only specific users should have access to that particular tenant network. Having the ability to enforce this via the SDN and demonstrate this is the case with an easy to understand SDN policy makes the network and security team's jobs easier as they have a real-time connectivity matrix for each application.

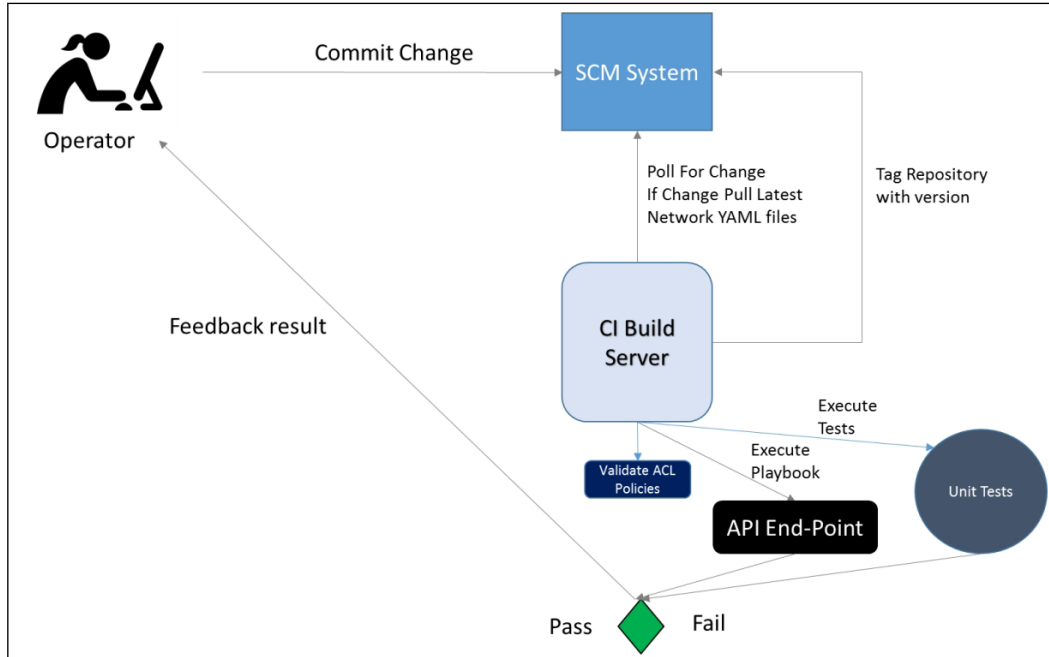
Wrapping security checks into continuous integration

Security checks should ideally be built into continuous integration processes, a concept covered in depth in *Chapter 7, Using Continuous Integration Builds for Network Configuration*. Otherwise, security teams would not be able to keep up with the daily changes being made to dynamic overlay networks and ever-changing network policies.

Compliance can be integrated with continuous integration processes by disallowing an allow-all policy when applied by a developer on their self-service ACL file for an application.

When a user commits this change to a source control management system, the **CI Build Server** starts a new continuous integration build. A validation on the continuous integration build for the SDN configuration build could be set up by the security team to reject this configuration and provide instant feedback to the user, as this breaks compliance.

The user would instead have to alter the self-service ACL policy rules to be implicit, so compliance then becomes just another validation of the continuous integration process, as shown here:



This is opposed to security teams auditing the ACL rules as a separate manual check, which would of course let ACL rules that breach security policy, slip through into production environments and allow attackers the potential to compromise a particular application, as its ACL rules are too open. This validation could even be done prior to a **CI Build Server** by running a simple Git hook, which would reject the commit after detecting an allow-all on the ACL policies by parsing the YAML file.

Using Cloud metadata

The use of cloud metadata is commonplace in public and private clouds such as AWS, Microsoft Azure, Google Cloud, and OpenStack as well as other cloud providers.

Tagging boxes with specific metadata has a variety of different use cases, and a subset of those use cases could greatly benefit a network or security team when dealing with particular network security challenges.

Cloud metadata, as covered already in this book, is a series of key-value pairs that are applied to a cloud server. If we take the example of a security vulnerability such as shell shock, which caused a series of DoS attacks when exploited in 2014, it is important that security vulnerabilities such as these are fixed immediately, to prevent attackers exploiting Linux boxes.

Within the remits of Continuous Delivery, it is important to make sure that if an issue occurs, then the mean time to recover is quick.

Take the scenario of vulnerability scanning. Each week, the whole overlay and underlay network will be scanned on a daily, or at worst, a weekly basis, using a security scanner.

Every time that the weekly network security scan runs on all boxes, it generates a report documenting a list of vulnerabilities for each server. This is subsequently reviewed by the service owners, and the security team will recommend specific patches or remediation over a number of days, so the mean time to resolve is high if important vulnerabilities are highlighted.

If instead of generating a separate report, the network security scan tagged the servers with a specific list of vulnerability IDs on their cloud metadata, then a complete inventory of vulnerabilities for the whole network would be available that could be acted upon to make real-time updates.

Using OpenStack as an example, the following command line could be executed to set metadata against a server when vulnerabilities are detected by using the `qualys_vul_ids` key value pair:

```
nova meta-data (instance-uuid) set qualys_vul_ids (qualys_id_list)
```

The following example would be executed as part of a script that would be run against all servers:

```
nova meta-data 061e8820-3abf-4151-83c8-13408923eb16 set qualys_vul_ids 23,122
```

This key value pair is then passed to the OpenStack metadata service, which will tag the OpenStack instance with all the relevant vulnerabilities that have been discovered as part of the **Qualys** vulnerability scan.

This will result in the OpenStack instance containing the following metadata:

Instance Overview	
Info	Meta
ID 061e8820-3abf-4151-83c8-13408923eb16	Key Name thoughtworks
Status Active	qualys_vul_ids 23,122
Availability Zone Prod	group riemann_prod
Created Oct. 9, 2015, 11:02 a.m.	hostname riemann.Prod.betfair
Uptime 2 days, 13 hours	runlist recipe[riemann::default]
	build 48

If a vulnerability such as shell shock was exposed by the security scan, then the network and security team could identify all servers with that vulnerability. In this case, Qualys ID 122 relates to shell shock, and targets the servers affected with an immediate patch.

Ansible dynamic inventory could be used to target the vulnerable boxes using a bespoke `ad_hoc_patch.yml` playbook with a when condition only, which executes patch commands to Linux servers if `Qualys ID 122` is tagged on the `qualys_vul_ids` metadata tag on the server.

The `ad_hoc_patch.yml` playbook would have the following steps to set a fact from the metadata and execute the commands only when the metadata tag contains the correct metadata:

```
- set_fact:
    metadata_tag: "{{ openstack.metadata.qualys_vul_ids }}"
- command: /usr/bin/yum clean all
  when: "122 in metadata_tag"
- yum: name=bash state=latest
  when: "122 in metadata_tag"
```

This playbook can be used to fix the shell shock **Bashdoor** bug immediately by executing following command:

```
ansible-playbook -i inventories/openstack.py -l Prod playbooks/ad_hoc_patch.yml
```

which would execute the playbook against all customer-facing servers in the `Prod` availability zone that contain the vulnerability, so target only production servers.

The playbook would only execute against servers in the production availability zone that match the metadata value of `122` as an active vulnerability using `Ansible jinja2` when filters, which would allow infrastructure engineers to remove the vulnerability in minutes. Imagine if security scanners did this metadata tagging as a feature of their scanner; it would help security massively.

Cloud metadata has many other use cases such as using an owner metadata tag on servers to send targeted e-mails or alerts if security teams detect any suspicious activity, or flag servers for re-deployment to install new patches when using immutable infrastructure.

Compromised servers can also be tagged as **quarantined** using metadata by security monitoring tools. Put simply, metadata allows teams to set server profiles using metadata, so a variety of actions can be carried out on them.

If a server is tagged as quarantined, a trigger could be set up to power down the server and migrate it to a quarantined micro-subnet in the Overlay network with no external access. This would allow a security team to carry out root cause analysis to ascertain how the box was compromised and mitigate the attack.



The important point to note is all these security processes can be automated to help maximize the features provided by public and private clouds. They should be looked upon as tools that can help automate and facilitate security processes rather than inhibit security.

Summary

In this chapter, we have looked at network security and ways in which security practices need to evolve to meet the demands of modern software-defined networks, as the industry has started to move away from flat layer 2 networks and instead utilize virtualized overlay networks.

This chapter has also hopefully debunked some of the fear and uncertainty associated with securing software-defined networks, while tackling hot topics such as the separation of test and production environments and the use of virtual firewalling for micro-segmentation as opposed to physical firewalls.

The focus of the chapter then shifted to strategies that can be adopted above and beyond minimum security requirements and looked at ways to secure SDN controllers and minimize the attack vectors. This can be achieved by isolating networks, creating out of band networks for network devices, appropriate authentication, and using TLS for inter-network device communication.

The chapter has also looked at the gains brought by implementing software-defined networking, such as the transparency and auditability of application to application connectivity. It has also explored opportunities to automate compliance checks by utilizing continuous integration best practices to validate ACL policies as part of continuous integration builds, rather than being a completely separate process. It has also explored leveraging cloud metadata to carry out emergency patching as opposed to it being a manual overhead, and covered other use cases for using cloud metadata such as quarantining servers and sending security notifications to teams.

This chapter brings us to the end of the book, which has looked at applying DevOps and Continuous Delivery principles to networking. The book has hopefully showed readers that networking does not need to be a manual set of tasks that slow down the whole Continuous Delivery process.

This book has covered a wide variety of topics that should hopefully give some food for thought and ideas that can be taken and implemented to improve network operations. Network automation is still relatively sparse in industry, but it doesn't need to be; the same automation principles that were applied to development, infrastructure, and testing are equally applicable to network operations.

Network teams shouldn't settle or accept the status quo, instead, be bold and, initiate real cultural change, and help improve network operations in the industry by embracing change and learning new skills.

More information:

- **Blog:** <http://devarmstrongops.blogspot.co.uk/>
- **LinkedIn:** <https://uk.linkedin.com/in/steven-armstrong-918629b1>
- **What is a Software-defined Network:** https://www.youtube.com/watch?v=1PL_oQT9tmc
- **SDN Fundamentals:** <https://www.youtube.com/watch?v=Np4p1CDIuzc>
- **SDN and OpenFlow:** <https://www.youtube.com/watch?v=1-DcbQhFAQs>

Index

Symbols

`_command` module 112, 113
`_config` module 113
`_template` module 114

A

Access Control List (ACL) 308

ACL rule

Egress 56
Ingress 56

Active Directory Domain

Services (ADDS) 308

agile testing

benefits, over siloed waterfall approach 229

Amazon Machine Image (AMI) 289

Amazon Web Services (AWS)

approach, for networking 16
availability zones 20
Elastic Load Balancing (ELB) 20
IP addressing 19
overview 12, 13
regions 20
security groups 19, 20
Virtual Private Cloud (VPC) 16-19

Ansible

about 99, 100, 308
`_command` module 112, 113
`_config` module 113
`_template` module 114
A/B subnets, storing in YAML files 181-183
ACL rules, storing in YAML files 181-183
core modules, used for network operations 111

for network operations 110

delegation 142, 143

directory structure 100, 101

dynamic inventory 147

inventory file 101

jinja2 filters 148, 149

jinja2 templates 106, 107

metadata, tagging 147, 148

modules 102, 103

networking modules, creating 149, 150

playbook, executing 104-106

roles 103, 104

serial, used for controlling roll

percentages 143-146

references 95

used, for configuring network

devices 107, 108

used, for orchestrating load balancers 142

used, for orchestrating SDN

controllers 178, 179

var files 106, 107

Ansible Control Host 99

Ansible Controller Node

inventories folder 101

library folder 101

playbooks folder 101

roles folder 101

Ansible Galaxy

about 109, 110

URL 108

Ansible network automation

reference 122

Ant

URL 189

Apache Mesos
URL 281
application connectivity topology 320
architectural components, Test Kitchens
 Busser framework
 driver 238
 platform 238
 provisioner 238
 suites 239
Arista
 references 122
Arista CloudVision platform 317
Arista CVX platform 318
Arista EOS
 reference 109
Arista EOS operating system 98, 99
Artifactory 266, 316
artifact repositories
 about 266
 Artifactory 266-268
Artifact Repository 251, 255, 259, 260, 265
Avi Controller 132
Avi Networks
 about 132
 reference 127

B

bash 86
Bashdoor bug 324
Block I/O (blkio) 285
blue green deployment process
 about 139-141
 release 1.1, deploying 140
 release 1.2, deploying 140
Border Gateway Protocol (BGP) 311
bottom-up DevOps initiatives
 complex problem, automating
 with network team 91, 92
 evangelizing, in network team 88, 89
 for networking teams 88
 sponsorship, seeking from respected
 manager/engineer 90
Bower
 URL 268
Business as Usual (BAU) 162
Business Process Management (BPM) 82

C

CD pipeline scheduler
 about 265-268
 Jenkins 269-272
centralized load balancing 124, 125
Chef
 URL 253
Chef Client 259
Chef Server 259
CI Build
 Server 188, 210, 233, 212, 235, 321, 322
Cisco
 references 122
Cisco IOS operating system 96, 97
Citrix NetScaler
 about 127-130
 reference, for products 128
cloud approaches
 about 1, 2
 hybrid cloud 4
 private cloud 3
 public cloud 2, 3
 software-defined operational 4
Cloudbase
 URL 10
Cloud Foundry 14
Cloud metadata
 using 322-324
CloudVision eXchange (CVX) 98
CloudVision eXchange (CVX) servers 317
command-line interface (CLI) 96
Commit Change 186
Compile Code 251
Completely Fair Scheduler (CFS) 285
Component Test Environment 259, 260
component testing 217
configuration management processes
 change requests 119
 desired state 115-118
 self-service operations 119, 120
 used, for managing network devices 114
configuration management processes,
 executed roles
 bgp 116, 117
 bridging role 116, 117
 common role 116, 117

- ecmp 117
- interface role 116, 117
- ipv4 role 116, 117
- mlag 117
- Connection tracking (conntrack) 158**
- Consul 287**
- containerization 282**
- containers**
 - about 279-281
 - benefits 285, 286
 - default Docker networking 292, 293
 - deploying 286
 - Docker 288
 - Docker daemon 288
 - Docker registry 288
 - Docker Swarm 294
 - Docker workflow 291
 - impact, on networking 303
 - Kubernetes 298
 - Linux control groups 285
 - Linux namespace 283, 284
 - packaging 289
 - Solaris Zones 282
 - user-defined bridge network, Docker 293
- containers, deploying**
 - CoreOS 287
 - etcd 287
- containers, packaging**
 - Dockerfile 289
 - Packer-Docker integration 289-291
- Continuous Delivery 161, 254-256, 294, 319**
- continuous integration build artifacts**
 - packaging, best practices 252
- continuous integration build**
 - servers 199, 200
 - used, for network devices 205, 206
 - used, for network orchestration 211, 212
- continuous integration (CI) 250**
 - about 161, 186, 187
 - database continuous
 - integration 190-193
 - developer continuous integration 188, 189
 - for network devices 210
 - package management 253
 - security checks, wrapping into 321, 322
 - tools 194
- continuous integration package**
 - management 250-252
- continuous integration testing 231-233**
- control groups (cgroups) 285**
- Control & Management Cluster level 11**
- CoreOS 287**
- CoreOS Rocket**
 - URL 281
- Cruise Control 200**

D

- database continuous integration 190-193**
- Debian**
 - URL 268
- declarative configuration sections, HAProxy**
 - Access Control List (ACL) 137
 - backend 136
 - check 136
 - frontend 137
- default Docker networking 292, 293**
- delegation 142, 143**
- Denial of Service (DoS) 315**
- continuous deployment 254-257**
- deployment artifacts**
 - packaging 262-264
- deployment methodologies**
 - pull model 258, 259
 - push model 260
 - push model/pull model, selecting 261, 262
- deployment pipelines**
 - network changes, deploying with 273
 - tools 265, 266
- developer continuous integration 188, 189**
- DevOps**
 - evangelizing, in network team 88
 - implementing, for networking team 77, 78
 - implementing, reasons 75, 76
 - initiating 74
 - references 93
- DevSecOps 309**
- distributed load balancing 124-126**
- Docker**
 - about 280, 288
 - URL 268, 281
 - user-defined bridge network 293

- Docker Compose 295
- Docker daemon 288
- Dockerfile 289, 298
- Docker machine 294
- Docker registry 288
- Docker Swarm
 - about 294
 - architecture 296, 297
- Docker workflow 291
- Domain Name System (DNS) 18
- Dynamic Host Configuration Protocol (DHCP) 19
- dynamic inventory 147

E

- Egress policies 311
- AWS elastic load balancing
 - reference 127
- Elastic Load Balancing (ELB) 20
- Electric Flow Deploy
 - URL 268
- endurance testing 219
- entities, Avi Networks
 - analytics profile 133
 - app profile 133
 - custom policy 133
 - health monitor profile 132
 - PKI profile 133
 - policy set 133
 - pool 133
 - SSL profile 133
 - TCP/UDP profile 133
 - virtual service 133
- entities, Citrix NetScaler
 - csvserver 130
 - gslbserver 130
 - gslbservice 130
 - gslbvserver 130
 - lbvserver 130
 - monitor 129
 - server 129
 - service 130
 - service group 130
- environment file 252
- Equal Cost Multipath (ECMP) 243

- Extensible Messaging and Presence Protocol (XMPP) 43
- external Border Gate Protocol (eBGP) 8

F

- F5 Big-IP
 - about 130, 131
 - Big-IP DNS 131
 - iRules 131
 - local traffic manager 131
 - monitor 131
 - pool 131
 - pool member 131
 - rate classes 132
 - reference 127
 - traffic class 132
 - virtual server 131
- failover testing 244
- feature branches 233
- Federal Information Processing Standard (FIPS) 128

G

- gated builds 233
- Generic
 - URL 268
- Git
 - about 101
 - reference 101
- Git LFS
 - URL 267
- Global Server Load Balancing (GSLB) 128
- Google Kubernetes
 - URL 281
- Gradle
 - URL 267
- Graphical User Interface (GUI) 2

H

- HAProxy
 - about 127, 135-137
 - URL 135
- HashiCorp 289
- Highly Available (HA) 45

hybrid cloud 4
Hyper-V 10

I

IBM Bluemix
URL 281

Identity and Access Management (IAM) service 17

immutable infrastructure
load balancing 137

immutable networking, software-defined networking
A/B immutable networking 174-176
application decommissioning 177
redundant firewall rules, cleaning 176, 177

immutable servers 138, 139

Information Technology Infrastructure Library (ITIL) 13

Infrastructure as a Service (IaaS)
function 14

Ingress policies 311

Integration Test Environment 259

integration testing 217

internal Border Gate Protocol (iBGP) 8

Internet Control Message Protocol (ICMP) 316

interprocess communication (IPC) 284

IP Address Management (IPAM) 310

iPerf tool 243

Ivy
URL 267

J

Jenkins 269, 308

jinja2 filters 148, 149

jinja2 templates 106, 107

Juniper
references 122

Juniper Junos operating system 98

K

Key Performance Indicators (KPIs) 221

knife 259

Kubelet 300

Kubernetes
about 298
architecture 298
kubectl 300, 301
SDN integration 302
worker node 299, 300

Kubernetes, architecture
master node 299

Kubernetes cluster
components 298

Kubernetes kubectl 300, 301

Kubernetes master node 299

Kubernetes SDN integration 302

Kubernetes worker node 299, 300

L

Layer 3 (L3) agents 23

Leaf-Spine networking architecture
advantages 9
implementing 7-9
Open vSwitch database (OVSDB) 9-11
versus, Spanning Tree Protocol (STP) 5

Lightweight Directory Access Protocol (LDAP) 23, 195, 308

Linux control groups 285

Linux namespace 283, 284

Load-Balancer-as-a-Service (LBaaS) 37

load balancing solutions
about 126, 127
Avi Networks 132
Citrix NetScaler 127-130
F5 Big-IP 130
HAProxy 136, 137
Nginx 133-135

load testing 219

LXC containers 97

M

makefile
tutorial, URL 189

Man in the Middle (MitM) 317

Maven
URL , 189

microservice architectures 123

Microsoft Azure Nano Server
URL 282
Modular Layer 2 (ML2) agents 23
MsBuild
URL 189
Multi-chassis Link Aggregation (MLAG) mode 8
Multipath Border Gate Protocol (MP-BGP) 43

N

Network Address Translation-Traversal (NAT-T) 16
network changes
deploying, with deployment pipelines 273
network changes deployment
configuration management tooling, incorporating 275
Continuous Delivery pipelines, network teams role in 276
deployment pipeline, steps 273, 274
failing fast 276, 277
feedback loops 276, 277
network self-service 273
network checklist 241, 242
network code quality tooling 244-246
network continuous integration
about 201, 202
continuous integration builds, used network devices 205, 206
network validation engines 203, 204
simple Jenkins network CI build, configuring 206-209
network continuous integration builds
validations, adding 209
network devices
configuring, Ansible used 107, 108
continuous integration builds, using for 205, 206
managing, configuration management processes used 114, 115
Network Function Virtualization (NFV) 16
networking
containers, impact on 303
networking teams
bottom-up DevOps initiatives 88

top-down DevOps initiatives 79
Network Interface Controller (NIC) 8
network operations, software-defined networking
API-driven networking, responsibilities 164
overlay architecture setup 164-170
self-service networking 171-173
network orchestration
continuous integration builds, used 211, 212
network security
about 319
account management 308-310
application connectivity topology 320
Cloud metadata, using 322-324
evolution 307, 308
firewalling 311
network device configuration 310
network segmentation 312-314
security checks, wrapping into Continuous Integration (CI) 321, 322
vulnerability detection 311, 312
network testing
assigning, to quality gates 236
network user journey 242, 243
network validation engines 203, 204
network vendors
Arista EOS operating system 98, 99
Cisco Ios operating systems 96, 97
Juniper Junos operating system 98
Nxos operating systems 96, 97
operating systems 96
Nexus 316
Nginx
about 127, 133-135
URL 133
nova scheduler rules 35
NPM
URL 267
Nuage Networks Virtual Service Platform (VSP) 313
Nuage VRS 46
Nuage VSD 46
Nuage VSPK object tree
building 179
Nuage VSP object model 165, 171

Nuage VSP platform

- brownfield projects, setting up 62-67
- greenfield projects, setting up 62-68
- multicast traffic, routing 68-70
- Nuage VSP software-defined
 - object model 49
- used, for integrating OpenStack 44-46
- Virtualized Service Controller (VSC) 42
- Virtualized Service Directory (VSD) 41
- Virtual Routing and Switching (VRS) 42

Nuage VSP software-defined object model

- layer 3 domain 49
- layer 3 domain template 49
- organization 49
- overview 49-61
- zone segments 50

NuGet

- URL 267

Nxos operating system 96, 97

O

OpenFlow 10, 313

OpenStack

- approach, for networking 21
- availability zones 35
- distributions 15
- instance, provisioning workflow used 36
- integrating, Nuage VSP platform
 - used 44-46
- LBaaS plugins 37
- Load-Balancer-as-a-Service (LBaaS) 37
- networks, provisioning 24-34
- neutron 23, 24
- Nuage VSD-managed mode 47, 48
- OpenStack-managed mode 47, 48
- overview 14, 15
- regions 35
- services 22
- tenant 23
- using, for Test Kitchen example 239-241

OpenStack Cinder

- reference 139

OpenStack platform 318

OpenStack services

- references 22

Open vSwitch 9

Open vSwitch Database (OVSDB)

- about 9, 313
- implementing 10, 11

Oracle Solaris Zones

- URL 281

Out Of Band (OoB) network 317

overlay networks 315

OVSDB 318

ovsdb-server 11

ovs-vswitchd daemon 11

P

PaaS solutions 125

Package 251

Packer

- about 289
- architecture 290

Peer review process 82

performance testing 219

Pivotal 14

Platform as a Service (PaaS) 4, 280

Plethora 308

post-processors 290

PowerShell 86

private cloud 3

Process ID (PID) 284

public cloud

- about 2, 3, 12
- Amazon Web Services (AWS),
 - overview 12, 13
- OpenStack, overview 14, 15

Public Key Infrastructure (PKI) 310

pull model

- about 258, 259
- example 258

Puppet

- URL 253

push model

- about 260
- example 260

push model/pull model

- selecting 261, 262

PyEZ 98

PyPi

- URL 268

Q

QEMU Copy On Write (QCOW) 289

quality assurance

best practices 227

best practices, applying to
networking 234-236

challenges 228

continuous integration testing 231-233

gated builds on branches 233

testing feedback loops, creating 230

quality gates

network testing, assigning, to 236

Quality of Service (QoS) 11, 131, 243

Qualys vulnerability scan 323

quarantined compromised servers 324

R

RabbitMQ 36

Rackspace Catrina

URL 281

Raft algorithm 287

Rake

URL 189

rakefile 208

Real-Time Scheduler (RTS) 285

Remote Procedure Call (RPC) 23

REST API 178

architecture 46

calls 46

Restriction of Hazardous

Substances (RoHS) 128

Rocket 280

Role Based Access Control (RBAC) 316

RSpec 238

S

Salt

URL 253

scalability testing 220

SCM System 188, 210, 212

SDN Controllers

orchestrating, Ansible used 178, 179

Secure Copy (SCP) 108

security checks

wrapping, into Continuous

Integration (CI) 321, 322

security groups 20

Selenium 242

Selenium test sample

URL 242

Service Level Agreements (SLA) 2

service-oriented architectures (SOA) 281

services, OpenStack

cinder 22

galera 22

glance 22

heat 22

horizon 22

ironic 22

keystone 22

neutron 22

nova 22

rabbitmq 22

swift 22

shadow IT 2

simple Jenkins network CI build

configuring 206-209

Smart System Upgrade (SSU) 99

**Software-defined Networking (SDN) 78,
308**

Continuous Delivery 161

Nuage, working 41-43

SDN controller, attacks 318

added network complexity 155, 156

agility 160

arguments 154, 155

benefits 159

complex networks, simplifying 162

controllers 40

immutable networking 174

mean time to recover 160

network operations, splitting up 162, 163

overlay, attacks 315, 316

performance 159

precision 160

redundancy 159

repeatability 160

scalability 159

securing 314

- skills, lacking 156, 157
- solutions 39-41
- stateful firewalling, lacking 158
- underlay, attacks 316, 317
- using, for disaster recovery 179, 180

software-defined operational 4

Solaris Zones 282

SonarQube

- architecture components 244

SonarQube Runner 245

Source Control Management (SCM) system 227

source control management (SCM) systems

- about 186, 194
- branching strategies 197-199
- centralized SCM systems 195, 196
- centralized SCM systems, examples 196
- distributed SCM systems 197
- distributed SCM systems, examples 197
- features 200

Spanning Tree Protocol (STP) 5

- implementing 5
- Open vSwitch database (OVSDb) 9-11
- versus, Leaf-Spine networking architecture 5

spike testing 219

stateful firewalls 310

static infrastructure

- load balancing 137, 138

static servers 138, 139

stress testing 219

subject matter expert (SME) 119

Sysdb 98

syslog server 319

system testing 218

T

Terminal Access Controller Access Control System (TACACS) 308

testing

- component testing 217
- integration testing 217
- performance testing 219
- relevance, to network teams 221, 222
- system testing 218

- unit testing 216
- user acceptance testing 220

testing feedback loops

- creating 230

Test Kitchen

- about 238
- example, with OpenStack 239-241

Test Kitchens Busser framework 238

test tools

- about 238
- failover testing 244
- network checklist 241, 242
- network code quality tooling 246
- network user journey 242, 243
- Quality of Service (QoS) 243
- unit testing tools 238

ThoughtWorks 308

Time To Live (TTL) 19

Tool Command Language (TCL) 131

tools, continuous integration (CI)

- about 194
- continuous integration build servers 199, 201
- source control management (SCM) systems 194

tools, deployment pipeline

- artifact repositories 266
- CD pipeline scheduler 268

top-down DevOps initiatives

- activity diagrams, mapping out 81-83
- behavior, changing of network teams 86, 87
- for networking teams 79
- operational model, changing of network team 84-86
- successful teams, analyzing 79, 80

Total Cost of Ownership (TCO) 3

Transactions Per Second (TPS) 128

Transmission Control Protocol (TCP) 8

Transport Layer Security (TLS) 310, 318

U

underlay network 316

unit testing 216

unit testing tools 238

Unit Tests 251

Urban Code Deploy
URL 268
user acceptance testing 220
User layer, OSI model 154

V

Vagrant
URL 268
Virtual Extensible LAN (VXLAN) 11, 311
Virtual IP (VIP) 37, 43, 126
Virtualized Service Controller (VSC) 42
Virtualized Service Directory (VSD) 41
Virtualized Service Gateway (VSG) 43, 62
Virtual Private Cloud (VPC) 12, 16-19
Virtual Routing and Switching (VRS) 42
Virtual Tunnel End Points (VTEPs) 313
V-Model
about 223
structure 223-227
VMware Photon
URL 282
volume testing 220
VXLAN Tunnel Endpoint (VTEP) 11

W

waterfall processes 222

X

XL Deploy
URL 268

Y

YAML files
A/B subnets, storing in 181-183
ACL rules, storing in 181-183
YUM
URL 268

Z

Zero Touch Provisioning (ZTP) 99
Zero Touch Replacement (ZTR) 99

