



Containers in Cisco IOS-XE, IOS-XR, and NX-OS

Orchestration and Operation

ciscopress.com

Yogesh Ramdoss
Nagendra Kumar Nainar

FREE SAMPLE CHAPTER
SHARE WITH OTHERS



Quotes for Praise

The world of networking continues to advance, and the emerging possibilities enabled by advancements like Kubernetes and Linux containers make keeping current on best practices a necessity to fully maximize Cisco IOS-XE, IOS-XR, and NX-OS platforms. I find this book to be an exceptionally insightful guide to understanding configuring, activating, orchestrating, and developing operational best practices for containerizing and instantiating applications and network services. I recommend it highly, as it is written by the engineers who have the best real-world experience of designing and troubleshooting these architectures.

—*Tom Berghoff, Senior Vice President, Customer Experience, Cisco Systems*

Time-to-value realization for our customers is key to achieving their business outcomes with our solutions. The book that Yogesh and Nagendra have written will allow our customers to maximize their investment by leveraging Cisco solutions in a unique and innovative way. The passion that Yogesh and Nagendra bring to this topic through this book will jump off the page at you. Please enjoy this well-written book, as it will help to ensure that you realize maximum value from the investments you have made in this Cisco solution.

—*Marc Holloman, Vice President, Customer Experience, Cisco Systems*

With the introduction of software-defined networking (SDN), we knew that the way we build and operate networks was never going to be the same. For the network to keep up with the speed of business, major changes had to happen at every layer of the enterprise stack. With this book, the reader will get an insider look at the innerworkings of the technologies enabling this change. Two experts in the field present solutions and technologies for successful virtualization and orchestration of network resources and applications on Cisco platforms. A must-read for every network technology participant in the digital transformation journey.

—*Hazim Dabir, Distinguished Engineer, Customer Experience, Cisco Systems*

At last, a comprehensive overview of containerization in networking context. Written by two of the foremost experts, this book is a must for every cloud architect and network architect contemplating the usage of containerized apps on Cisco routers for on-board computing-related use cases such as efficient network operations.

—*Rajiv Asati, CTO/Distinguished Engineer, Customer Experience, Cisco Systems*

To digitize and disrupt legacy business models, agile organizations require a flexible and scalable infrastructure. Containers provide portability, protection, and design resiliency to modern network infrastructures. Learn to leverage the power of containers on the Cisco Systems platforms from two of its leading minds. As authors, patent developers, and principle engineers at Cisco, Nagendra and Yogesh are uniquely qualified to explain how you, too, can build, test, deploy, and manage application hosting on your Cisco-enabled infrastructure. I rely on their expertise daily and, through this book, you will, too!

—*Chris Berriman, Senior Director, Customer Experience, Cisco Systems* and author of *Networking Technologies, Fundamentals of Network Management* and *Cisco Network Management Solutions*

As society becomes more dependent on the benefits of the digital economy, enterprise leaders need to be informed of the technological capabilities available to meet their customers' demand. In this book, two of the most prolific principal engineers and technologists at Cisco, Nagendra Kumar Nainar and Yogesh Ramdoss, provide a clear introduction into one of the major innovations fundamental to the digital economy: cloud computing and virtualization. The work is superb in content, covering the basics of container orchestration and networking from a technically agnostic perspective while progressing to show the unique capabilities and benefits of using Cisco technology. For the leader responsible for digital transformation, this book will provide actionable insights through real-world scenarios and a future look to the possibilities ahead.

—*Hector Acevedo, Senior Director, Customer Experience, Cisco Systems*

Containers in Cisco IOS-XE, IOS-XR, and NX-OS: Orchestration and Operation

Yogesh Ramdoss (CCIE No. 16183)
Nagendra Kumar Nainar
(CCIE No. 20987, CCDE No. 20190014)

Cisco Press
Hoboken, New Jersey

Containers in Cisco IOS-XE, IOS-XR, and NX-OS: Orchestration and Operation

Yogesh Ramdoss (CCIE No. 16183)

Nagendra Kumar Nainar (CCIE No. 20987, CCDE No. 20190014)

Copyright© 2021 Cisco Systems, Inc.

Published by:
Cisco Press

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

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

ScoutAutomatedPrintCode

Library of Congress Control Number: 2020906738

ISBN-13: 978-0-13-589575-7

ISBN-10: 0-13-589575-8

Warning and Disclaimer

This book is designed to provide information about Cisco containers. Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied.

The information is provided on an “as is” basis. The authors, Cisco Press, and Cisco Systems, Inc. shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the discs or programs that may accompany it.

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

Trademark Acknowledgments

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

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Feedback Information

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

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

We greatly appreciate your assistance.

Editor-in-Chief: Mark Taub

Alliances Manager, Cisco Press: Arezou Gol

Director, ITP Product Management: Brett Bartow

Executive Editor: Nancy Davis

Managing Editor: Sandra Schroeder

Development Editor: Rick Kughen

Project Editor: Mandie Frank

Copy Editor: Gill Editorial Services

Technical Editors: Richard Furr; Rahul Nimbalkar

Editorial Assistant: Cindy Teeters

Designer: Chuti Prasertsith

Composition: codeMantra

Indexer: Erika Millen

Proofreader: Charlotte Kughen



Americas Headquarters
Cisco Systems, Inc.
San Jose, CA

Asia Pacific Headquarters
Cisco Systems (USA) Pte. Ltd.
Singapore

Europe Headquarters
Cisco Systems International BV
Amsterdam, The Netherlands

Cisco has more than 200 offices worldwide. Addresses, phone numbers, and fax numbers are listed on the Cisco Website at www.cisco.com/go/offices.

Cisco and the Cisco logo are trademarks or registered trademarks of Cisco and/or its affiliates in the U.S. and other countries. To view a list of Cisco trademarks, go to this URL: www.cisco.com/go/trademarks. Third party trademarks mentioned are the property of their respective owners. The use of the word partner does not imply a partnership relationship between Cisco and any other company. (1110R)

Credits

Figure 1-5 Screenshot of Linux processor output © Canonical Ltd

“cloud computing is defined as a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources that can rapidly provisioned and released with minimal management effort or service provider interaction.” NIST

About the Authors

Yogesh Ramdoss (CCIE No. 16183) is a principal engineer with the Cisco Customer Experience (CX) organization focusing on data center technologies such as Nexus switching platforms (standalone as well as VXLAN fabric), application-centric infrastructure (ACI), and hyperconverged infrastructure HyperFlex. Associated with Cisco since 2003, Yogesh is a distinguished speaker at Cisco Live, where he shares his knowledge and educates customers and partners on data center platforms and technologies, telemetry, analytics, network programmability, and various troubleshooting and packet capturing tools. He is a machine and behavior learning coinventor.

Nagendra Kumar Nainar (CCIE No. 20987, CCDE No. 20190014) is a principal engineer with the Cisco Customer Experience (CX) organization (formerly TAC), focusing on enterprise networking. He is the coinventor of more than 100 patent applications on various cutting-edge technologies and the coarchitect for various recent technologies. He has coauthored multiple Internet RFCs and IEEE papers. Serving as Technical Program Committee (TPC) member for various IEEE and other international conferences, he is an active speaker in various industry forums.

About the Technical Reviewers

Richard Furr, CCIE No. 9173 (R&S & SP), is a technical leader of the Cisco Customer Experience (CX) organization, providing support for customers and TAC teams around the world. Richard has authored and acted as a technical editor for several Cisco Press publications. During the past 19 years, Richard has provided support to service provider, enterprise, and data center environments resolving complex problems with routing protocols, MPLS, IP Multicast, IPv6, and QoS.

Rahul Nimbalkar is a technical leader with Cisco, where he has worked since 2001. He has been working on LXC (guestshell, OAC) and Docker containers as well as virtualization technologies on the Cisco NX-OS data center switches. Most recently he designed and implemented Docker container support on the standalone NX-OS 9000 series and has been working on the NX-OSv, the virtual machine version of the NX-OS switch.

Dedications

Yogesh: I dedicate this book to my parents, Ramdoss Rajagopal and Bhavani Ramdoss, who have given me the best things in the world and taught me to do the right things to the people around us. I further dedicate this book to my wife, Vaishnavi, and our children, Janani and Karthik, for their patience and support throughout the authoring process. Finally, I want to dedicate this book to my ex-manager Mike Stallings, for several years of his support and encouragement to start this project.

Nagendra: This book is dedicated to my mother and father for their encouragement and support throughout my life. I am what I am today because of the guidance and support from you. This book is further dedicated to my wife, Lavanya, and my daughter, Ananyaa, who are the driving factors of my life. Lavanya, your patience and understanding are a great support for me to do things beyond my capacity. Ananyaa, I love you to the core. This book is also dedicated to my ex-manager, Mike Stallings. Mike, you are one of the greatest managers, and I am glad that I had an opportunity to work with you.

Acknowledgments

Yogesh: My heartfelt thanks to my manager Hector Acevedo for his trust and support. I am thankful to my coauthor, Nagendra Kumar Nainar, for his guidance, and to my technical reviewers, Richard Furr and Rahul Nimbalkar, for providing valuable comments and feedback. I would like to extend my thanks to Christopher Hart for helping me build, test, and validate deployment scenarios and use cases for this book. Last but not least, I would like to thank Carlos Pignataro, who has been mentoring me and providing career guidance for so many years.

Nagendra: First, I would like to thank my mentor, Carlos Pignataro, for his mentoring and career guidance. Your guidance and advice always played a key role in my career. I would like to thank my manager, Chris Berriman, for his support and flexibility. Your trust in me and the flexibility you offer is encouraging, and it motivates me to explore different opportunities.

I would like to thank my coauthor and good friend, Yogesh Ramdoss, who completed this book on time. I would like to thank Richard Furr and Rahul Nimbalkar for the high-quality technical review performed on our writeup. You improved the quality of the content.

I also would like to thank Akshar Sharma, who helped me significantly with various XR content, and Akram Sheriff, who helped with IoT-related content.

I would like to thank Ajitha Buvanachandran for the tremendous help in reviewing my work. I would like to thank Poornima Nandakumar and Satish Manchana for helping me with various virtualization use cases.

Contents at a Glance

Foreword xxv

Introduction xxvii

Part I Virtualization and Containers

Chapter 1 Introduction to Virtualization 1

Chapter 2 Virtualization and Cisco 23

Chapter 3 Container Orchestration and Management 61

Chapter 4 Container Networking Concepts 97

Part II Container Deployment and Operation in Cisco Products

Chapter 5 Container Orchestration in Cisco IOS-XE Platforms 139

Chapter 6 Container Orchestration in Cisco IOS-XR Platforms 189

Chapter 7 Container Orchestration in Cisco NX-OS Platforms 235

Chapter 8 Application Developers' Tools and Resources 291

Chapter 9 Container Deployment Use Cases 361

Chapter 10 Current NFV Offering and Future Trends in Containers 405

Index 425

Contents

	Foreword	xxv
	Introduction	xxvii
Part I	Virtualization and Containers	
Chapter 1	Introduction to Virtualization	1
	History of Computer Evolution	1
	History of Virtualization	2
	Motivation and Business Drivers for Virtualization	3
	<i>Resource Optimization</i>	4
	<i>Resilience</i>	5
	<i>Simplicity and Cost Optimization</i>	5
	Virtualization—Architecture Definition and Types	6
	Architecture and Components	6
	Types of Virtualization	8
	<i>Server Virtualization</i>	8
	<i>Network Virtualization</i>	10
	<i>Storage Virtualization</i>	12
	Connecting the Dots with Cloud Computing	13
	Computing Virtualization Elements and Techniques	14
	Virtual Machines	14
	Containers	15
	Serverless Computing	17
	Virtualization Scale and Design Consideration	18
	High Availability	18
	Workload Distribution	19
	<i>Resource Utilization</i>	19
	Multitenancy in Virtualization	19
	Summary	20
	References in This Chapter	21
Chapter 2	Virtualization and Cisco	23
	History of Virtualization in Cisco	23
	Network Infrastructure Virtualization	23
	Network Device Virtualization	26
	Virtualization in Enterprise and Service Provider Environments	30
	Enterprise	30
	Service Provider	31

The Era of Software-Defined Networking	32
SDN Enablers	33
Control Plane Virtualization	33
SDN Controllers	34
<i>OpenFlow</i>	34
<i>Open Source Controllers</i>	35
APIs and Programmability	36
API	36
Programmability	38
Cisco Proprietary SDN Controllers	42
<i>APIC</i>	42
<i>APIC-EM</i>	44
<i>DNA Center</i>	45
<i>Modern Network Design with SDN and NFV</i>	47
Elements in Network Function Virtualization	48
Orchestration and Deployment of Virtual Network Services	48
Technology Trends Built on SDN	51
Internet of Things (IoT)	51
<i>Cisco's IoT Platform for Industries</i>	52
The Cisco IoT Platform for Service Providers	53
<i>A Use Case for IoT with SDN: Manufacturing</i>	55
Intent-Based Networking (IBN)	57
Summary	58
References in This Chapter	59
Chapter 3	Container Orchestration and Management 61
Introduction to the Cloud-Native Reference Model	61
Application Development Framework	62
Automated Orchestration and Management	62
Container Runtime and Provisioning	63
The Journey from Virtual Network Function (VNF) to Cloud Native Function (CNF)	63
Container Deployment and Orchestration Overview	65
Linux Containers (LXC)	66
<i>Cisco Service Containers</i>	67
<i>Cisco Application Hosting Framework</i>	69
<i>Cisco Guest Shell</i>	70

Cisco Open Agent Containers 71

Docker 75

Kubernetes 79

Container Deployment and Orchestration 81

Orchestrating and Managing Containers Using LXC 81

Orchestrating and Managing Containers Using Docker 84

Docker Daemon Status Verification 85

Docker Client 86

Getting Docker Images 87

Running the Container 89

Orchestrating and Managing Containers Using Kubernetes 91

Running Docker Daemon 91

Enabling Kubernetes Master 92

Enabling Nexus 9000 Switch as Kubernetes Worker Node 93

Deploying Workload Using Kubernetes 94

Summary 95

References 95

Chapter 4 Container Networking Concepts 97

Container Networking—Introduction and Essentials 97

Application to Host 98

Application to Application 98

Application to External Network 98

Container Networking 99

Namespace to External Network 100

Namespace to Namespace 102

Key Points 104

Container Network Models and Interfaces 105

Cisco Native App Hosting Network Model 106

Shared Network Mode 106

Dedicated Network Mode 108

Docker Networking—Container Network Model 111

None Networking 113

Host Networking 113

Bridge Networking 114

Overlay Networking 114

Macvlan 114

Kubernetes Container Network Interface (CNI) Model	114
Setting Up Container Networking	115
Native App Hosting—Shared Networking Configuration	115
<i>Cisco IOS-XE Configuration</i>	115
<i>Cisco IOS-XR Configuration</i>	117
<i>Cisco Nexus OS Configuration</i>	122
<i>Support Matrix</i>	125
Native App Hosting—Dedicated Networking Configuration	125
Cisco IOS XE Configuration	125
<i>Routing Mode—Numbered</i>	126
<i>Routing Mode—Unnumbered</i>	128
<i>Layer 2 Mode</i>	129
<i>Cisco IOS XR and Nexus OS</i>	131
Docker Network Configuration	131
<i>None Networking</i>	131
<i>Host Network</i>	132
<i>Bridge Networking</i>	134
Kubernetes	136
Summary	136
References	137

Part II Container Deployment and Operation in Cisco Products

Chapter 5 Container Orchestration in Cisco IOS-XE Platforms 139

Cisco IOS-XE Architecture	139
Brief History of IOS-XE	140
Architecture Components and Functions	141
<i>Switching Platforms</i>	142
<i>Routing Platforms</i>	144
IOS-XE Architecture: Application Hosting	146
libvirt and Virtualization Manager	146
IOx Overview	148
IOx Applications	149
Application Types	150
Resource Requirements	153
<i>Memory and Storage Requirements</i>	153
<i>VirtualPortGroup</i>	154
<i>Virtual NIC (vNIC)</i>	155

Application Deployment Workflow and Operation States	156
Developing and Hosting Applications	157
LXC-Based Guest Shell Container	157
<i>Activating IOx</i>	157
<i>Setting Up Network Configuration</i>	157
<i>Activating the Guest Shell Container</i>	159
Developing PaaS-Style Applications and Hosting	161
<i>Supported Platforms</i>	161
<i>Setting Up the Development Environment</i>	161
<i>Developing a Python Application</i>	161
<i>Creating a Docker Image</i>	162
<i>Creating an IOx Package Using YAML</i>	162
<i>Installing, Activating, and Running the Application</i>	165
Developing Virtual Machine–Based Application and Hosting	166
<i>Setting Up an Application Development Environment</i>	167
Building the Virtual Machine File System	169
<i>Build an IOx Package Using YAML</i>	170
<i>Installing, Activating, and Running the Application</i>	172
Developing and Hosting a Docker-Style Application	175
<i>Setting Up Docker Toolchain</i>	175
<i>Caveats and Restrictions</i>	176
<i>Development Workflow</i>	177
<i>Images and Package Repository</i>	177
Develop Python Application	178
Build Docker Image	179
<i>Building an IOx Application Package Using YAML</i>	180
<i>Installing, Activating, and Running the Application</i>	182
Native Docker Application Hosting in Catalyst 9300	182
<i>Workflow 1: Building and Exporting a Docker Image</i>	182
<i>Workflow 2: Performing a Docker Pull and Export</i>	184
Deploying Native Docker Applications	184
<i>Docker Container Networking</i>	185
Licensing Requirements	185
Summary	186
References	187

Chapter 6 Container Orchestration in Cisco IOS-XR Platforms 189

Cisco IOS-XR Architecture	189
Architecture and Software Evolution	190
Application Hosting Architecture	192
Kernel Interface Module (KIM)	193
Network Namespaces	195
Docker Hosting Architecture	196
Hosting Environment Readiness	198
Storage	198
CPU Share	199
Memory	200
Types of Application Hosting in Cisco XR Platform	201
Native Application Hosting	201
Native Hosting from an Existing RPM File	202
Building an RPM File for Native Hosting	206
LXC-Based Application Hosting	209
Network Configuration and Verification	216
Docker-Based Application Hosting	217
Docker Images and Registry	218
Loading from Public Registry	218
Loading from Local Registry	220
Loading Manually to Local Store	222
Container Deployment Workflow	223
Network Configuration and Verification	224
Network Reachability Configuration	224
Name Resolution Configuration	224
Network Proxy Configuration	225
Application Hosting in VRF Namespace	226
VRF Namespace	226
Application Hosting in VRF Namespace Using LXC	229
Container Management	232
Persistent Application Deployment	232
Summary	234
References	234

Chapter 7 Container Orchestration in Cisco NX-OS Platforms 235

Cisco NX-OS Software Architecture	235
NX-OS Foundation	235

NX-OS Modular Software Architecture	236
Fault Detection and Recovery	237
More Benefits of NX-OS	238
Hosting Environment Readiness	239
Guest Shell	239
<i>Platforms Support</i>	239
<i>Platform Resource Requirements</i>	240
Bash	240
LXC-based Open Agent Container (OAC)	240
<i>Platforms Supported</i>	241
<i>Platform Resource Requirements</i>	241
Container Infrastructure Configuration and Instantiation	242
Guest Shell	242
<i>Guest Shell OVA File</i>	242
<i>Deployment Model and Workflow</i>	243
<i>Accessing Guest Shell</i>	245
<i>Accessing Guest Shell via SSH</i>	246
<i>Guest Shell Networking Setup and Verification</i>	248
<i>Installation and Verification of Applications</i>	253
<i>Custom Python Application</i>	253
<i>Python API-Based Application</i>	254
Bash	256
<i>Enabling Bash</i>	256
<i>Accessing Bash from NX-OS</i>	257
<i>Accessing Bash via SSH</i>	258
Docker Containers	260
<i>Docker Client</i>	261
<i>Docker Host</i>	261
<i>Starting Docker Daemon</i>	263
<i>Instantiating a Docker Container with Alpine Image</i>	263
<i>Managing Docker Container</i>	266
<i>Orchestrating Docker Containers Using Kubernetes</i>	268
<i>Orchestrating Docker Containers in a Node from the K8s Master</i>	273
Open Agent Container (OAC)	276
<i>OAC Deployment Model and Workflow</i>	277
<i>Accessing OAC via the Console</i>	280

	<i>OAC Networking Setup and Verification</i>	280
	<i>Management and Orchestration of OAC</i>	284
	<i>Installation and Verification of Applications</i>	285
	<i>Custom Python Application</i>	285
	<i>Application Using Python APIs</i>	287
	<i>Package Management</i>	288
	Summary	288
	References	289
Chapter 8	Application Developers' Tools and Resources	291
	Cisco Development Tool Kits and Resources	291
	Nexus Software Development Kit (NX-SDK)	291
	<i>NX-SDK Release Versions</i>	292
	<i>NX-SDK Deployment Modes</i>	293
	<i>NX-SDK Installation and Activation</i>	293
	Python APIs—IOS-XE / NX-OS	297
	<i>Python API in NX-OS</i>	297
	<i>Python API in IOS-XE</i>	302
	Nexus API (NX-API)	305
	<i>Transport</i>	306
	<i>Message Formats</i>	306
	<i>Security</i>	306
	<i>Enabling NX-API</i>	306
	<i>Data Management Engine and Managed Objects</i>	309
	<i>NX-API REST</i>	310
	RESTCONF, NETCONF, and YANG	318
	Enabling RESTCONF Agent in IOS-XE	320
	Using a RESTCONF Agent in IOS-XE	321
	Enabling RESTCONF Agent in NX-OS	323
	Using the RESTCONF Agent in NX-OS	325
	Enabling NETCONF Agent in IOS-XE	327
	Using the NETCONF Agent in IOS-XE	329
	Enabling NETCONF Agent in IOS-XR	331
	Using NETCONF Agent in IOS-XR	332
	Enabling the NETCONF Agent in NX-OS	333
	Using NETCONF Agent in NX-OS	333

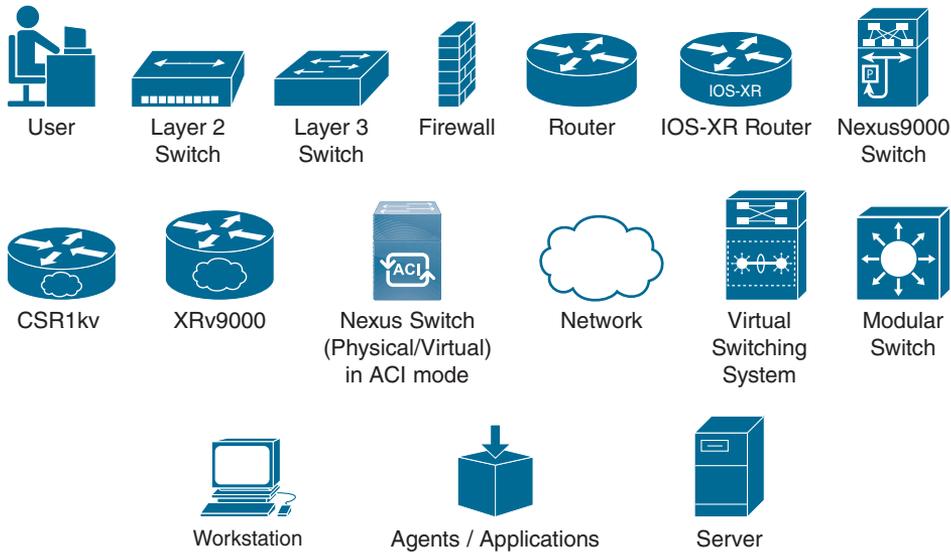
Open-Source and Commercial Tools	336
Linux	336
Apache NetBeans	337
GitHub	337
Atom	338
AWS Cloud9	338
Zend Studio	339
Eclipse	339
Bootstrap	340
Bitbucket	340
Node.js	341
Building and Deploying Container Images	341
Build Docker Images	341
<i>Dockerfile</i>	342
<i>Docker Build</i>	343
<i>Docker Run</i>	343
Publish Docker Images—Docker Hub	344
<i>Docker Hub Account</i>	344
<i>Docker Hub Repository</i>	344
<i>Docker Hub Publish</i>	344
<i>Docker Pull</i>	345
<i>Docker Registry</i>	345
Configuration and Application Management Tools	345
Ansible	346
Puppet	346
Chef	346
Ansible and IOS-XE	347
<i>Hosts File</i>	347
<i>Authentication</i>	348
<i>Sample Playbook</i>	348
<i>Running a Playbook</i>	349
<i>NETCONF Operations with Ansible</i>	350
<i>Puppet and NX-OS</i>	351
<i>Installing and Activating the Puppet Agent</i>	351
<i>Using Puppet Agent</i>	353
<i>Chef and IOS-XR</i>	354

	<i>Creating a Chef Cookbook with Recipes</i>	354
	<i>Installing and Activating Chef Client</i>	355
	Summary	357
	References	357
Chapter 9	Container Deployment Use Cases	361
	General Use Cases for Enterprise, Service Provider, and Data Center Networks	362
	Inventory Management	362
	Hardware and Software Stability Check	362
	Control Plane Health Check	362
	Resource Usage and Scalability Check	362
	Configuration Consistency Check	362
	Traffic Profiling and Top Talkers	363
	Monitor Operational Data to Detect Failures	363
	Build Infrastructure for Proof-of-Concept and Testing Purposes	363
	Create and Deploy DHCP Docker Container	363
	<i>Configure the Catalyst Switch for Application Hosting</i>	363
	<i>Create Docker Containers</i>	365
	<i>Install and Activate DHCP Docker Container in Catalyst 9000</i>	368
	Create and Deploy DNS Docker Container	369
	<i>Prepare to Create DNS Docker Container</i>	370
	<i>Create DNS Docker Containers</i>	373
	<i>Install and Activate DNS Docker Container in Catalyst 9000</i>	374
	Create HAProxy and Node Containers	375
	<i>Project Initiation</i>	375
	<i>Setting Up Web Server</i>	376
	<i>Create Docker Image</i>	377
	<i>Deploy, Install, and Activate Web Server Docker Containers</i>	378
	<i>HAProxy Load Balancer Setup</i>	380
	<i>Create Docker Image</i>	381
	<i>Install, Activate, and Run HAProxy Docker Containers</i>	382
	IOS-XR Use Case: Disaggregated Seamless BFD as a Virtual Network Function for Rapid Failure Detection	384
	Seamless BFD Overview	385
	S-BFD Discriminator	386
	S-BFD Reflector Session	386

Creating and Hosting S-BFD as a Virtual Network Function	387
S-BFD Docker Images	388
Hosting the S-BFD Reflectorbase on the XR Device	388
Hosting the S-BFD Client on the Server	390
NX-OS Use Case: Control Plane Health Check Using an Anomaly Detector	391
Objective of the Application	391
Build and Host the Anomaly Detector Application in Docker—High-Level Procedure	392
Floodlight Application	392
<i>Capturing Traffic</i>	394
<i>Classifying Expected and Unexpected Control Plane Traffic</i>	395
Running the App in NX-OS	396
NX-OS Use Case: NX-OS Docker Health Check	398
Objective of the Application	398
Build and Host the Application in Docker—High-Level Procedure	398
NX-OS Docker Health Check Application	399
<i>Performing Health Check</i>	399
Running the App in NX-OS	401
Summary	404
Chapter 10 Current NFV Offering and Future Trends in Containers	405
App Hosting Services	405
Solenoid	406
Two-Way Active Measurement Protocol (TWAMP)	407
tcpdump	407
Cisco Kinetic EFM Module	408
perfSONAR	408
DNS/DHCP	409
NetBeez Agent	409
App Hosting Summary	410
Cisco NFV Offerings	411
Compute Platforms	412
<i>Cisco Unified Computing Servers (UCS)</i>	412
ENCS	412
Virtual Routers and Switches	414
Cisco Ultra Service Platform	415

Cisco Container Platforms	416
Consolidated View	417
Containers and Service Chaining	418
Network Service Header	419
Segment Routing	420
Serverless Computing and Network Functions	421
Summary	423
References	423
Index	425

Icons Used in This Book



Command Syntax Conventions

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

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

Foreword

What do rhinos, Internet RFCs, memory buffers, and volunteering have in common? In that intersection, you will find Nagendra and Yogesh!

A few years ago, Nagendra, Yogesh, and I met when we were providing direct customer support on networking infrastructure. It was the common passion for technology as well as for customer service that brought us together.

At that time, virtual local area networks (VLANs), then followed by encapsulations or “tunnels,” were network virtualization technologies. Fast-forwarding, we can see how the technology landscape and ecosystem has dramatically evolved and reinvented itself. Virtualization has multiplied and expanded into many areas of networking, storage, and computing, including containerized network functions.

This book guides and shepherds the reader through the evolution of virtualization technologies with depth and breath. It invites images both historical and futuristically visionary. Explaining design principles and considerations of end-to-end network virtualization and controller architectures, it challenges the reader through the mystical twists of context virtualization, recursing through a software stack as if it were a Möbius strip and creating dimensional layers on a topology.

Not content with abstraction alone, this book teaches the concrete art and engineering of configuring container orchestration and management—where the network is the node, and the node is the network. It describes how to create bendable cloud-native network functions, which are utilized for lifecycle management and service automation. Then, to make Cloud real, this book instructs network wizards and software geeks how to realize these container-based orchestration architectures in Cisco network operating system software and platforms. Because the end goal is to conjure applications that were not possible before, this book covers several application developer tools, resources, and real-life container deployment use cases.

Switching contexts—no pun intended—what do technology and helping others have in common? Everything! And as technology reinvented itself, so did the focus area and the meaning of their careers.

Yogesh not only masters all interface counters in Nexus fabric platforms, but bends packet captures and embedded diagnostic tools and configures Layer 2 (L2) Data Center Interconnect (DCI). He also applies his skills and experience to design and deploy networks, ad honorem, for the Food Bank of Central and Eastern North Carolina. He leads selflessly.

Nagendra’s technical and architectural accomplishments include coinventing several patented applications—in fact, he is currently the top inventor in the Cisco Customer Experience team—and coauthoring Internet RFC standards. I’m honored to have partnered in white-boarding some of those with him. However, what makes me prouder is that together we painted houses for refugee resettlement. When I asked Nagendra for help in a hackathon in Bangalore, India, for the Cisco Sustainable Impact program, he

jumped at the chance. For techno-conservation including saving rhinos and other endangered species, and for encouraging social responsibility, he engaged immediately with no second thought.

The seeds of the technology that the two authors are showing in this book might have always been there, but it needed thought leaders to reinvent it to accommodate its fast-paced growth. Similarly, Nagendra and Yogesh, throughout the years, have reinvented themselves as technologists and as leaders, continually growing. Professional portability is reached with technology-agnostic technical skills, as they evidence from VLANs to containers. And they further extrapolated that to food banks and techno-conservation.

I am grateful to Yogesh and Nagendra for contributing to the industry and sharing in writing this book on container orchestration!

Carlos Pignataro

Distinguished Engineer and CTO, Emerging Technologies and Incubation, Cisco Systems

Adjunct Faculty, North Carolina State University

Fellow, National Academy of Inventors

Volunteer

Introduction

The introduction of cloud computing and virtualization is one of the radical innovations that the industry has witnessed recently. These technologies have allowed the industry to decouple the services from the proprietary hardware and allowed the users to instantiate workflows on any supported compute platforms. Although toolsets such as Linux Containers (LXC) and Kernel Virtual Machine (KVM) were developed to instantiate any workloads as virtual machines, recent developments, such as Docker and Kubernetes, allow the user to develop and instantiate these workloads as containers. Containerizing the applications and network services (NFV) is the goal the industry is moving toward for the agility and efficiency properties.

Cisco IOS-XE, IOS-XR, and NX-OS Architecture have been augmented with compute virtualization capabilities to accommodate native and third-party container hosting that allows the users to containerize and instantiate applications or network services. The Software Development Kit (SDK) Cisco offers can be used to develop applications from scratch to instantiate on Cisco IOS-XE, IOS-XR, and NX-OS platforms natively by leveraging the built-in application hosting capabilities.

This book explains the architecture and capabilities of Cisco products, Container infrastructure configuration, activation, orchestration, and operational activities. It acts as a complete guide to deploying and operating applications and network services that are hosted on Cisco platforms. This book is the first and the only comprehensive guide featuring Cisco IOS-XE, IOS-XR, and NX-OS architecture that supports deployment of various virtual and containerized network services and the container orchestration tools to instantiate and operate them.

Goals and Methods

The primary goal of this book is to introduce you to the new application hosting capabilities and the built-in toolkits that can be used to build, orchestrate, and operate applications or services by leveraging compute resources in Cisco platforms.

This book introduces readers to the fundamentals of virtualization and associated concepts and how virtualization and SDN are related to the Cisco IOS-XE, IOS-XR, and NX-OS platforms. This book explores different orchestration tools (for example, LXC, KVM, Docker, and Kubernetes) for workload instantiation (as virtual machines or containers) and different modes of enabling the interworkload communication. It takes a deep dive into application hosting capabilities for each of the mentioned Cisco platforms. Furthermore, it covers available Cisco and open-source tools and resources that application developers can leverage to build and test applications before hosting them on the respective platforms. Beyond explaining the platform capabilities and the methods to host applications, this book offers multiple real-world use cases in which these applications are used in day-to-day network operations.

How This Book Is Organized

Although you could read this book cover to cover, it is designed to be flexible and allow you to easily move between chapters and sections of chapters to cover just the material you need, when you need it.

Part I, “Virtualization and Containers,” is an overview of the evolution of virtualization technologies and different orchestration tools and networking concepts that are broadly applicable for hosting a virtual service in any compute platform.

- Chapter 1, “Introduction to Virtualization”: This chapter starts by describing the evolution of computing technologies and then introduces the motivation, business drivers, and concept of computing virtualization. It also describes the architecture, principles, and various types of virtualization.
- Chapter 2, “Virtualization and Cisco”: This chapter describes the history of virtualization in the Cisco core routing and switching products and discusses how infrastructure virtualization is being achieved in these platforms. This chapter continues to introduce the software-defined networking (SDN) concepts, associated protocols, Cisco and open-source controllers, function virtualization, and trending technologies.
- Chapter 3, “Container Orchestration and Management”: This chapter describes the cloud-native reference model and how the model is used to develop cloud-native services and help the industry migrate from virtual to cloud-native network functions. It explains different orchestration tools and the applicability of these tools for workload instantiation on Cisco platforms.
- Chapter 4, “Container Networking Concepts”: This chapter describes the fundamentals of container networking and how the underlying kernels use the network namespaces to create resource isolation. This chapter digs deep into different container networking models for each orchestration method and explains all the supported modes in Cisco platforms along with the relevant configuration to enable the container networking modes.

Part II, “Container Deployment and Operation in Cisco Products,” discusses fundamentals of IOS-XE, IOS-XR, and NX-OS architecture; various container capabilities natively available in related platforms; and how to leverage them to host applications to perform day-to-day operations.

- Chapter 5, “Container Orchestration in Cisco IOS-XE Platforms”: This chapter starts with a quick introduction to the architecture of IOS-XE and its key components and functions. Next, it explains how the architecture enables application hosting with support for various types of applications. It offers sample steps to enable, install, activate, and orchestrate the containers (such as LXC) and how to leverage them to host simple applications.
- Chapter 6, “Container Orchestration in Cisco IOS-XR Platforms”: This chapter introduces the IOS-XR architecture and the latest enhancements to support

application hosting capabilities for native or third-party services. The chapter further explains different methods of hosting the application using orchestration tools, such as LXC and Docker, along with the relevant network configurations. This chapter concludes by explaining the basic management aspects of the hosted applications.

- Chapter 7: “Container Orchestration in Cisco NX-OS Platforms”: This chapter introduces users to the fundamentals of the NX-OS architecture and the benefits this architecture brings to hosting applications natively. It discusses various container capabilities, such as Guest Shell, Bash, Docker, and more, and covers the steps to activate and configure them and host applications. The chapter concludes with an explanation of how a Docker container running in a Nexus platform can be orchestrated with Kubernetes.
- Chapter 8: “Application Developers’ Tools and Resources”: In this chapter, you will learn various Cisco as well as open-source tools and resources available to application developers to develop, test, and host applications in Cisco IOS-XE, IOS-XR, and NX-OS platforms. It provides details on the software development environment and toolkits that are built into these platforms.
- Chapter 9: “Container Deployment Use Cases”: This chapter introduces various real-world use cases for Day-0, Day-1, and Day-2 operations and explains the applicability of the use cases with deployment examples on Cisco IOS-XR, IOS-XE, and NX-OS platforms.
- Chapter 10: “Current NFV Offering and Future Trends in Containers”: This chapter starts by introducing various open-source and certified third-party applications that are readily available for hosting on Cisco platforms for some common use cases. It continues by explaining different NFV services currently offered by Cisco and highlights some virtualization trends.

This page intentionally left blank

Container Orchestration in Cisco NX-OS Platforms

In this chapter, you will learn the following:

- Cisco NX-OS architecture—key characteristics and benefits
- Environment readiness to host containers and applications
- Container infrastructure instantiation, network and access configuration, orchestration and application hosting—LXC-based Guest Shell, Bash, Docker and LXC-based Open Agent Container.

Cisco NX-OS Software Architecture

Cisco NX-OS is designed to meet the needs of modern data centers, which demand products, applications, and services that are high performance, highly available, resilient, secure, scalable, and modular in architecture. These criteria are met by all the platforms—Nexus 3000, 5000, 6000, 7000, and 9000—that support and run Cisco NX-OS. These characteristics provide the solid foundation of resilience and robustness necessary for network device OSes powering the mission-critical environment of today's enterprise-class data centers.

NX-OS Foundation

Cisco NX-OS finds its roots in the Cisco SAN-OS operating system used in lossless SAN networks. As a direct result of having been deployed and evolving from nearly a decade in the extremely critical storage area networking space, NX-OS can deliver the performance, reliability, and lifecycle expected in the data center.

Cisco NX-OS is built on a Linux kernel. By using Linux kernel as its foundation, Cisco NX-OS has the following characteristics and benefits:

- An open-source and community development model, which leads to real-world field testing and rapid defect identification and resolution

- Proven stability and maturity, with advanced capabilities
- A near-real-time OS kernel, which is suitable to scale real-time applications
- An architecture leveraging multiple run-queues for handling multicore and multiple-CPU system configurations
- A multithreaded, preemptive multitasking capability that provides protected fair access to kernel and CPU resources because it employs a highly scalable processor queue and process-management architecture

These characteristics and benefits ensure system stability and fair access to the system resources for software functions such as routing protocols, the spanning tree, and internal services and processes. By its inherent nature, NX-OS supports multiprocessor and multicore hardware platforms, which help to simplify scalability by supporting not only current hardware or software features but also future software features.

NX-OS Modular Software Architecture

NX-OS software components are modular and built on top of the Linux kernel, as illustrated in Figure 7-1. These modular components can be described as such:

- Hardware drivers, which are hardware-related and highly dependent on the platform
- Infrastructure modules to manage the system
- Software features or control-plane functions

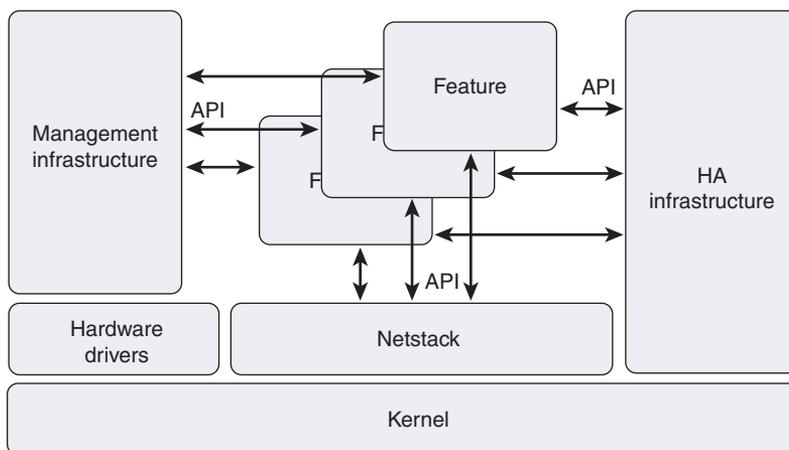


Figure 7-1 NX-OS Software Architecture

The platform-dependent modules consist of hardware-related subsystems, such as hardware and chipset drivers specific to a particular hardware model on which Cisco NX-OS runs. These modules typically provide standardized APIs and messaging capabilities to

upper-layer subsystems. The modules essentially constitute a hardware abstraction layer to enable consistent development at higher layers in the OS, improving overall OS portability. The code base for these hardware-dependent modules reduces the overall code that needs to be ported to support future NX-OS releases and for other hardware platforms.

The Netstack module runs in user space and is a complete TCP/IP stack with components L2 Packet Manager, ARP, Adjacency Manager, IPv4, Internet Control Message Protocol v4 (ICMPv4), IPv6, ICMPv6, TCP/UDP, and socket library. The Netstack is built and used to handle the traffic sent to and from the CPU. A user can debug Netstack to uncover the process(es) that are triggering a high CPU utilization condition.

The system infrastructure modules such as management infrastructure and high-availability infrastructure provide essential base system services that enable process management, fault detection and recovery, and interservice communication. High-availability infrastructure provides subsecond recovery of a fault, enabling stateful recovery of a process. During the recovery, it preserves the runtime state of the feature, increasing the overall network and services availability. The Persistent Storage System (PSS) and Message Transmission Services (MTS), the core parts of the high-availability infrastructure, enable the subsecond recovery of a fault, resulting in overall higher system uptime.

The feature modules consist of the actual underlying services responsible for delivering a feature or running a protocol at the control plane level. Open Shortest Path First (OSPF), Border Gateway Protocol (BGP), Spanning Tree Protocol, Overlay Transport Virtualization (OTV), and NetFlow export are all examples of modularized system-level features or protocols. Each feature is implemented as an independent, memory-protected process that is spawned as needed based on the overall system configuration.

This approach differs from that of legacy network operating systems in that only the specific features that are configured are automatically loaded and started. This highly granular approach to modularity enables benefits such as these:

- Compartmentalization of fault domains, resulting in overall system resiliency and stability
- Simplified portability for cross-platform consistency
- More efficient defect isolation, resulting in rapid defect resolution
- Easy integration of new feature modules into the OS
- Support of conditional services, resulting in efficient use of memory, CPU and CLI resources, and improved security as lesser OS functions are exposed

Fault Detection and Recovery

In addition to the resiliency gained from architectural improvements, Cisco NX-OS provides internal hierarchical and multilayered system fault detection and recovery mechanisms. No software system is completely immune to problems, so it is important to have an effective strategy for detecting and recovering from faults quickly, with as little effect as possible. Cisco NX-OS is designed from the start to provide this capability.

Individual service and feature processes are monitored and managed by the Cisco NX-OS System Manager, an intelligent monitoring service with integrated high-availability logic. The system manager can detect and correct a failure or lockup of any feature service within the system. The system manager is, in turn, monitored and managed for health by the Cisco NX-OS kernel. A specialized portion of the kernel is designed to detect failures and lockups of the Cisco NX-OS System Manager. The kernel itself is monitored through hardware. A hardware process constantly monitors the kernel health and activity. Any fault, failure, or lockup at the kernel level is detected by hardware and triggers a supervisor switchover. Figure 7-2 illustrates the components involved in the fault detection and recovery process.

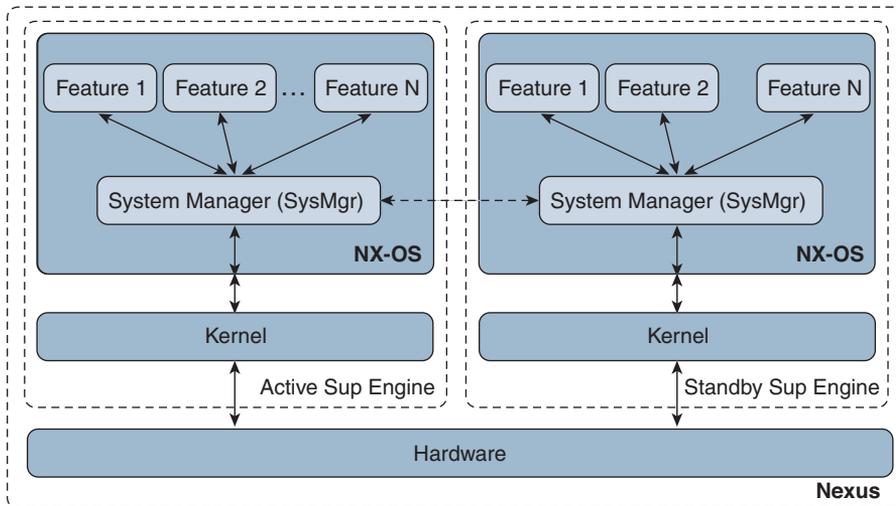


Figure 7-2 *NX-OS Fault Detection and Recovery*

The combination of these multilevel detection and health monitoring systems creates a robust and resilient operating environment that can reduce the overall effect of internal faults and, more importantly, preserve the stability of the overall network by internalizing these types of events.

More Benefits of NX-OS

Following are some of the key but nonexhaustive benefits that this chapter will briefly discuss:

- **Familiar usability and operation:** Cisco NX-OS maintains the familiarity of the Cisco IOS CLI. Users comfortable with the Cisco IOS CLI will find themselves equally comfortable with Cisco NX-OS, which has numerous user interface enhancements.

- **Virtualization capability:** Cisco NX-OS offers the capability to virtualize the platform on which it is running. Using Cisco NX-OS virtual device contexts (VDCs), a single physical device can be virtualized into many logical devices, each operating independently. And it supports virtualization of overlay transport, which addresses the need to scale Layer 2 by extending the domain across different data centers.
- **Enhanced security:** NX-OS supports features and tools to secure the platform and its functions. Some of the more advanced security features supported are Cisco TrustSec (CTS), IP Source Guard, DHCP snooping, Unicast Reverse Path Forwarding (uRPF), access control lists (ACLs), and 802.1x.
- **Unified I/O and unified fabric:** Cisco Unified Fabric includes the flexibility to run Fiber Channel; IP-based storage, such as network-attached storage (NAS) and Small Computer System Interface over IP (iSCSI), or FCoE; or a combination of these technologies on a converged network.
- **Support of standalone fabric:** NX-OS supports features to build standalone fabrics such as FabricPath, Dynamic Fabric Automation (DFA), and VXLAN/EVPN to scale the Layer 2 domains and meet the demands of today's virtualized computing environments and applications.
- **Advanced system management:** NX-OS supports SNMP (v1, v2c, and v3) to enable traditional ways of managing systems. NETCONF and XML are integrated to NX-OS, which make it IETF-compliant to transact XML through secure connections. With the support of configuration checkpoint and rollback, managing devices through its software lifecycle is easier.

Hosting Environment Readiness

This section discusses the various shells and containers supported in Nexus switching platforms and the OS version and resources required to support them.

Guest Shell

Guest Shell is an execution environment isolated from the host operating system's kernel space and running within a Linux Container (LXC). As with OAC, having a decoupled execution space allows customization of the Linux environment to suit the needs of the applications without affecting the host system or applications running in other Linux Containers.

Platforms Support

Guest Shell is supported in Nexus 3000/9000 platforms. Table 7-1 provides the minimum NX-OS version required for each platform to run the Guest Shell environment.

Table 7-1 *Nexus Switches and NX-OS Versions Supporting Guest Shell*

Platforms	Minimum Version
Nexus 3000 series	7.0(3)I2(1)
Nexus 9000 series	7.0(3)I2(1)

Platform Resource Requirements

The Guest Shell reserves a specific amount of memory in Bootflash. Upon activation, it reserves dynamic RAM and CPU resources, as shown in Table 7-2.

Table 7-2 *Nexus Resource Requirement for Guest Shell*

Platforms	DRAM Reservation	Bootflash Reservation	CPU reservation
Nexus 3000 series	256 MB	200 MB	1%
Nexus 9000 series	256 MB	200 MB	1%

By default, Nexus switches with 4 GB of RAM will not enable Guest Shell. Use the `guestshell enable` command to install and enable Guest Shell.

Bash

In addition to Guest Shell, Cisco Nexus9000 Series devices support access to the Bourne-Again Shell (Bash). Bash interprets commands that you enter or commands that are read from a shell script. The following sections discuss how Bash enables access to the underlying Linux system on the device and how it manages the system. Bash shell is supported on both Cisco Nexus 3000 series as well as 9000-series platforms, as shown in Table 7-3.

Table 7-3 *Nexus Switches and NX-OS Versions Supporting Bash*

Platforms	Minimum Version
Nexus 3000 series	6.1(2)I2(2)
Nexus 9000 series	6.1(2)I2(2)

The coming sections discuss how Bash enables direct and root access to the underlying kernel and how it instantiates the Docker service and containers.

LXC-based Open Agent Container (OAC)

OAC is a 32-bit, CentOS 6.7-based container that is built specifically to support open agents like Puppet and Chef to manage Nexus switching platforms.

With the current architecture, Open Agents cannot be directly installed and run on Nexus platforms. To overcome this challenge, a special environment is built, which is a decoupled execution space within an LXC called as the Open Agent Container (OAC). Having an execution space that is decoupled from the native host system enables customization of the environment to meet the applications' requirements without affecting the host systems' applications or any other containers.

Platforms Supported

Open Agent Container is one of the earliest container environments supported in Nexus platforms, and it is supported only in Nexus 5600, Nexus 6000, and Nexus 7000/7700 series platforms. Table 7-4 shows the minimum NX-OS release required for each platform supporting OAC.

Table 7-4 *Nexus Switches and NX-OS Versions Supporting OAC*

Platforms	Minimum Version
Nexus 5600 series	7.3(0)N1(1)
Nexus 6000 series	7.3(0)N1(1)
Nexus 7000/7700	7.3(0)D1(1)

Platform Resource Requirements

As the file required to instantiate and for associated data storage, OAC occupies up to a specific memory size in bootflash. Upon activation, it requires dynamic RAM and CPU resources, as shown in Table 7-5.

Table 7-5 *Nexus Resource Requirement for OAC*

Platforms	DRAM Reservation	Bootflash Reservation	CPU Reservation
Nexus 5600 series	256 MB	400 MB	1%
Nexus 6000 series	256 MB	400 MB	1%
Nexus 7000/7700	256 MB	400 MB	1%

Note: The OAC functionality will no longer be supported on the Nexus 7000 from 8.4.1 release onward. When executing the commands to enable OAC, users will be notified about the deprecation. Even though the feature is deprecated, this book covers OAC as a significant install base of Nexus 7000 that still runs pre-8.4.1 releases.

Container Infrastructure Configuration and Instantiation

This section explains and provides procedures to instantiate different types of containers and to access, configure, manage, and orchestrate them. It also provides detailed steps to deploy and manage applications in the containers.

Guest Shell

Just as with OAC, Cisco Nexus 3000/9000 Series devices support access to an isolated execution environment, called the Guest Shell, which is running within a secure Linux Container (sLXC), as illustrated in Figure 7-3. Under the hood, the Guest Shell is just a libvirt-managed LXC container. This Guest Shell is based on CentOS 7 and can be managed using traditional Linux commands.

The Guest Shell has various functions and offers key benefits that aid developers in building and hosting applications in Nexus platforms, such as providing access to the network, NX-OS CLI, bootflash filesystem, and above all, the ability to install Python scripts and Linux applications.

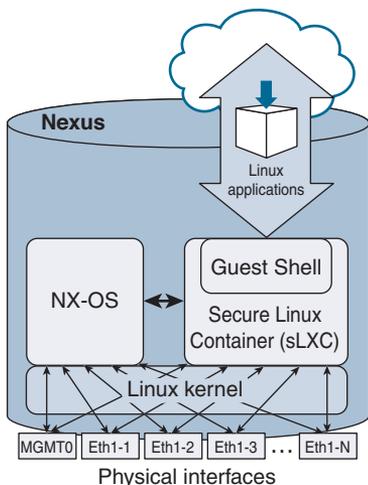


Figure 7-3 *Guest Shell in NX-OS*

Guest Shell OVA File

In Nexus 3000 and 9000 switches, the .ova file for the default version of Guest Shell is integrated with the NX-OS image, and as previously discussed, you do not have to download and install an .ova to enable it.

Deployment Model and Workflow

It is simple to activate Guest Shell in supported platforms, and it can be done with one command, as shown in Example 7-1. The Guest Shell needs to be explicitly activated only in the first generation of the Nexus 3000 platform that came with 4 GB RAM. In later generations of Nexus 3000 and Nexus 9000 platforms, Guest Shell is enabled by default.

This `guestshell enable` command does the following:

1. Creates a virtual service instance
2. Extracts the `.ova` file built into NX-OS
3. Validates the contents in the file
4. Creates a virtual environment in the device
5. Instantiates the Guest Shell container

Example 7-1 Enable Guest Shell in NX-OS

```
N3K-C3548P# guestshell enable
2019 Sep 12 02:04:00 N3K-C3548P %$ VDC-1 %$ %VMAN-2-INSTALL_STATE: Installing
virtual service 'guestshell+'
N3K-C3548P#
N3K-C3548P# show virtual-service list
Virtual Service List:
Name                Status              Package Name
-----
guestshell+        Activating          guestshell.ova
N3K-C3548P#
2019 Sep 12 02:04:55 N3K-C3548P %$ VDC-1 %$ %VMAN-2-ACTIVATION_STATE: Successfully
activated virtual service 'guestshell+'
N3K-C3548P# show virtual-service list
Virtual Service List:
Name                Status              Package Name
-----
guestshell+        Activated           guestshell.ova
N3K-C3548P#
```

To know the resources allocated to the shell, use the `show guestshell` command. As you can see in Example 7-2, it reports the operational state of the shell, disk, memory, and CPU resource reservation, and it reports the `filesystems/devices` mounted in the shell. The utilization command shown next shows usage of memory, CPU, and storage resources in real time.

Example 7-2 *Guest Shell Status and Resource Allocation*

```

N3K-C3548P#
N3K-C3548P# show guestshell
Virtual service guestshell+ detail
  State                : Activated
  Package information
    Name                : guestshell.ova
    Path                : /isanboot/bin/guestshell.ova
  Application
    Name                : GuestShell
    Installed version   : 2.4(0.0)
    Description         : Cisco Systems Guest Shell
  Signing
    Key type            : Cisco release key
    Method              : SHA-1
  Licensing
    Name                : None
    Version             : None
  Resource reservation
    Disk                : 250 MB
    Memory              : 256 MB
    CPU                 : 1% system CPU

  Attached devices
  Type                Name                Alias
  -----
  Disk                _rootfs
  Disk                /cisco/core
  Serial/shell
  Serial/aux
  Serial/Syslog       serial2
  Serial/Trace        serial3
N3K-C3548P#
N3K-C3548P# show virtual-service utilization name guestshell+
Virtual-Service Utilization:

CPU Utilization:
  Requested Application Utilization: 1 %
  Actual Application Utilization: 0 % (30 second average)
  CPU State: R : Running

Memory Utilization:
  Memory Allocation: 262144 KB
  Memory Used:      13444 KB

```

```

Storage Utilization:
  Name: _rootfs, Alias:
    Capacity(1K blocks): 243823      Used(1K blocks): 156896
    Available(1K blocks): 82331      Usage: 66 %

  Name: /cisco/core, Alias:
    Capacity(1K blocks): 2097152     Used(1K blocks): 0
    Available(1K blocks): 2097152    Usage: 0 %
N3K-C3548P#

```

By default, the resources allocated to the Guest Shell are small compared to the total resources available in a switch. An administrator can change the size of the CPU, memory, and root filesystem (rootfs) resources allocated to the Guest Shell by using **guestshell resize** commands in the configuration mode. Note that after changing resource allocations, a Guest Shell reboot is required. This can be achieved by using the **guestshell reboot** command, which basically deactivates and reactivates the Guest Shell.

Accessing Guest Shell

By default, the Guest Shell starts with an open-ssh service as soon as it is enabled. The server listens to TCP port 17700 on the local host loopback IP interface 1270.0.1. This provides password-less access to the Guest Shell from the NX-OS, as shown in Example 7-3.

Example 7-3 Access Guest Shell

```

N3K-C3548P#
N3K-C3548P# guestshell
[admin@guestshell ~]$
[admin@guestshell ~]$ whoami
admin
[admin@guestshell ~]$ hostnamectl
  Static hostname: guestshell
        Icon name: computer-container
        Chassis: container
        Machine ID: 2a79cdc74cdc45659ad7788742da0599
        Boot ID: 295a7ceda3684f3caa2d5597de8ae1e0
  Virtualization: lxc-libvirt
  Operating System: CentOS Linux 7 (Core)
        CPE OS Name: cpe:/o:centos:centos:7
        Kernel: Linux 4.1.21-WR8.0.0.25-standard
        Architecture: x86-64
[admin@guestshell ~]$
[admin@guestshell ~]$
[admin@guestshell ~]$ ps -ef | grep 17700

```

```

UID          PID  PPID  C  STIME TTY      TIME    CMD
root          91    1    0  Aug30 ?        00:00:00 /usr/sbin/sshd -D -f /etc/ssh/
             sshd_config-cisco -p 17700 -o ListenAddress=localhost
admin        1515  1495  0  18:40 pts/4    00:00:00 grep --color=auto 17700
[admin@guestshell ~]$

```

Notice that the file used to spawn the default SSH process is `/etc/ssh/sshd_config-cisco`. If this file is altered, the `guestshell` command might not function properly. If that occurs, it is recommended that you destroy and re-enable the Guest Shell.

Accessing Guest Shell via SSH

To access the Guest Shell, you need to be in the switch first and then access the shell using the `guestshell` command mentioned earlier in this chapter in “Accessing Guest Shell.” This access can be slow, and it is highly preferable to have a direct SSH access.

As you see in Example 7-4, after logging into the Guest Shell, check the SSH configuration—the TCP port it is listening to and the IPv4/v6 addresses associated to the SSH service. Because NX-OS has allocated TCP port number 22 to the SSH process running in the switch, configure an unused and different TCP port number for the Guest Shell’s SSH daemon. As you see in Example 7-4, `/etc/ssh/sshd_config` has Port 2222 assigned to the service, and it is listening for connections at 10.102.242.131, which is the IP address assigned to the Ethernet1/1 interface of the switch. Make sure to configure the DNS server for name resolution and domain information for the Guest Shell and the applications installed in it to resolve domain names.

Example 7-4 Guest Shell Networking

```

[admin@guestshell ~]$ more /etc/ssh/sshd_config
<snip>
Port 2222
#AddressFamily any
ListenAddress 10.102.242.131
#ListenAddress ::
<snip>
[admin@guestshell ~]$
[admin@guestshell ~]$ cat /etc/resolv.conf
nameserver 8.8.8.8
search example.com
[admin@guestshell ~]$

```

In any CentOS-based Linux platform, Guest Shell uses `systemd` as its service manager. Therefore, `systemctl` commands can be used to start, stop, restart, reload, or check the status of the SSH service, as shown in Example 7-5. Check the status of the SSH service before starting it.

Example 7-5 *Activate SSH Service*

```

[admin@guestshell etc]$ systemctl start sshd
[admin@guestshell ~]$
[admin@guestshell ~]$ systemctl status sshd.service
sshd.service - OpenSSH server daemon
Loaded: loaded (/usr/lib/systemd/system/sshd.service; disabled; vendor preset:
        enabled)
Active: inactive (dead)
<snip>
[admin@guestshell ~]$
[admin@guestshell ~]$ systemctl start sshd.service -l
[admin@guestshell ~]$
[admin@guestshell ~]$ systemctl status sshd.service -l
sshd.service - OpenSSH server daemon
Loaded: loaded (/usr/lib/systemd/system/sshd.service; disabled; vendor preset:
        enabled)
Active: active (running) since Sat 2019-08-31 15:33:52 UTC; 4s ago
Main PID: 886 (sshd)
        CGroup: /system.slice/sshd.service
                └─886 /usr/sbin/sshd -D
Aug 31 15:33:52 guestshell sshd[886]: Executing: /usr/sbin/sshd -D
Aug 31 15:33:52 guestshell sshd[886]: Server listening on 10.102.242.131 port 2222.
[admin@guestshell ~]$

```

As shown in Example 7-6, make sure the TCP socket assigned to Guest Shell's SSH service is open and in the listening state. Because Guest Shell uses *kstack* networking implementation, a Kernel Socket is allocated for TCP port 2222, as shown in Example 7-6.

Example 7-6 *Open Kernel Sockets in Nexus Switch*

```

N3K-C3548P#
N3K-C3548P# show sockets connection
Total number of netstack tcp sockets: 3
Active connections (including servers)

```

	Protocol	State/Context	Recv-Q/ Send-Q	Local Address(port) / Remote Address(port)
[host]:	tcp(4/6)	LISTEN	0	* (22)
		Wildcard	0	* (*)
[host]:	tcp	LISTEN	0	* (161)
		Wildcard	0	* (*)
[host]:	tcp(4/6)	LISTEN	0	* (161)
		Wildcard	0	* (*)

```
<snip>
Kernel Socket Connection:
Netid      State      Recv-Q    Send-Q    Local Address:Port    Peer Address:Port
tcp        LISTEN     0          128       10.102.242.131:2222   *:*
<snip>
```

Once the SSH service is up and running and all the configured sockets are in the listening state, users can access Guest Shell via SSH from an external device, as shown in Example 7-7.

Example 7-7 SSH Access to Guest Shell

```
root@Ubuntu-Server1$ ssh -p 2222 admin@10.102.242.131
admin@10.102.242.131's password:
Last login: Sat Aug 31 11:42:26 2019
[admin@guestshell ~]$
```

It is possible to run multiple instances of SSH Server daemons and associate them to any VRF active in the switch. In other words, the Guest Shell can be accessed via SSH through two sockets associated to different namespaces or VRFs, hence from different networks. Example 7-8 shows that the switch has two sockets open: one for **management** VRF and the other one for **default** VRF. The socket allocated for the SSH service in the **default** VRF is (172.16.1.1:5123) and is (10.102.242.131:2222) for the **management** VRF.

Example 7-8 SSH Service per Namespace

```
[admin@guestshell ~]$ chvrf default
[admin@guestshell ~]$
[admin@guestshell ~]$ /usr/sbin/sshd -p 5123 -o ListenAddress=172.16.1.1
[admin@guestshell ~]$
[admin@guestshell ~]$ exit
N3K-C3548P#
N3K-C3548P# show sockets connection | include Netid|2222|5123
Netid State      Recv-Q    Send-Q    Local Address:Port    Peer Address:Port
tcp    LISTEN     0          128       172.16.1.1:5123      *:*
tcp    LISTEN     0          128       10.102.242.131:2222  *:*
N3K-C3548P#
```

Guest Shell Networking Setup and Verification

Guest Shell is a powerful container and application hosting environment because it provides access to every front-panel port, VLAN SVIs, and port-channels in the device. Using the Cisco **kstack** implementation, all these interfaces are represented and available as network devices in the Linux kernel.

With the command shown in Example 7-9, check the VRFs that are visible to the Guest Shell container, where each VRF is a Kernel Network Namespace, as represented in the Linux kernel.

Example 7-9 Guest Shell Namespaces

```
[admin@guestshell ~]$ ip netns list
management
default
[admin@guestshell ~]$
```

Figure 7-4 illustrates that namespaces created for each of the VRFs and shows the interfaces associated to each of these VRFs.

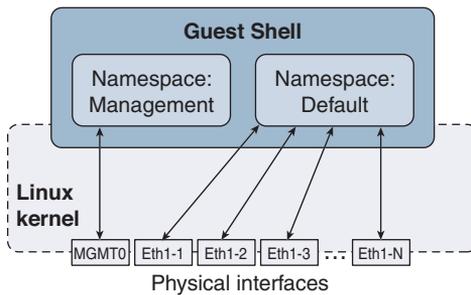


Figure 7-4 Guest Shell Namespaces

Because the physical and logical interfaces are accessible through network namespaces, the container can access network elements directly. As shown in Example 7-10, the `chvrf` command switches the context to a specific VRF, and `ifconfig -a` is used to list the interfaces associated to the current context.

The `chvrf` command is a helper utility that uses the `ip netns exec` command under the hood to switch the VRF context. Apart from the `ifconfig` command provided in this example, you can also use the `ip link show` command to obtain a list of interfaces associated to the specific context.

Example 7-10 Guest Shell Namespaces and Network Devices

```
[admin@guestshell ~]$ chvrf default
[admin@guestshell ~]$
[admin@guestshell ~]$ ifconfig -a
Eth1-1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.102.242.131 netmask 255.255.255.240 broadcast 10.102.242.143
    ether 00:3a:9c:5a:00:67 txqueuelen 100 (Ethernet)
    RX packets 2045299 bytes 469647600 (447.8 MiB)
    RX errors 0 dropped 1615524 overruns 0 frame 0
```

```

TX packets 556549 bytes 95536394 (91.1 MiB)
TX errors 0 dropped 892 overruns 0 carrier 0 collisions 0

Eth1-2: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
ether 00:3a:9c:5a:00:67 txqueuelen 100 (Ethernet)
RX packets 0 bytes 0 (0.0 B)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 0 bytes 0 (0.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

Eth1-3: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
ether 00:3a:9c:5a:00:67 txqueuelen 100 (Ethernet)
RX packets 0 bytes 0 (0.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

<snip>
Eth1-48: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
ether 00:3a:9c:5a:00:67 txqueuelen 100 (Ethernet)
RX packets 0 bytes 0 (0.0 B)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 0 bytes 0 (0.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

Lo100: flags=65<UP,RUNNING> mtu 1500
inet 10.1.1.1 netmask 255.255.255.0
ether 00:3a:9c:5a:00:60 txqueuelen 100 (Ethernet)
RX packets 0 bytes 0 (0.0 B)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 0 bytes 0 (0.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

<snip>
veobc: flags=67<UP,BROADCAST,RUNNING> mtu 1494
inet 127.1.2.1 netmask 255.255.255.0 broadcast 127.1.2.255
ether 00:00:00:00:01:01 txqueuelen 0 (Ethernet)
RX packets 0 bytes 0 (0.0 B)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 134 bytes 57112 (55.7 KiB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

<snip>
[admin@guestshell ~]$

```

All the software data structures, including ARP tables, routing tables, and prefixes, are synchronized between NX-OS and the Linux kernel by the NetBroker process, as illustrated in Figure 7-5. Because the Guest Shell uses the Linux `kstack`, the data structures synchronization is automatic.

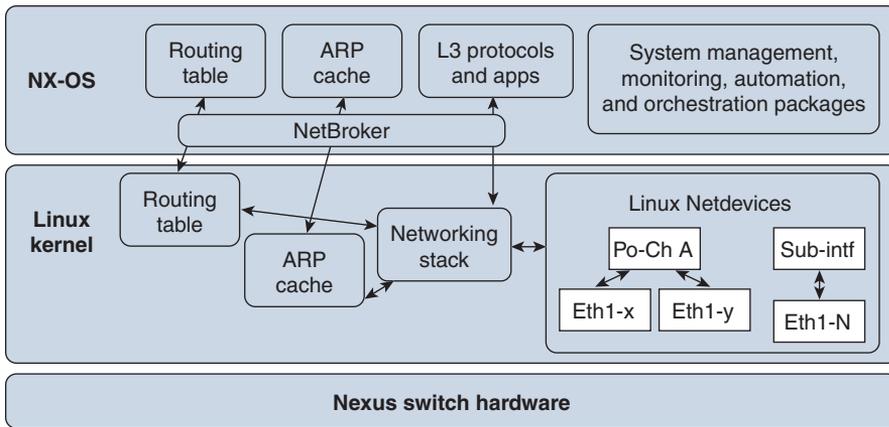


Figure 7-5 *NetBroker—Synchronize NX-OS and the Linux Kernel*

The commands provided in Example 7-11 show the routing table, interface configuration, and statistics as well as the ARP cache in a specific context.

Example 7-11 *Guest Shell Routing and ARP Tables—Default Namespace*

```
[admin@guestshell ~]$
[admin@guestshell ~]$ chvrf default route -vn
Kernel IP routing table
Destination      Gateway          Genmask          Flags Metric Ref  Use Iface
0.0.0.0          10.102.242.129  0.0.0.0          UG    51    0    0 Eth1-1
10.1.1.0         0.0.0.0         255.255.255.0   U     0     0    0 Lo100
10.102.242.128  0.0.0.0         255.255.255.240 U     0     0    0 Eth1-1
10.102.242.129  0.0.0.0         255.255.255.255 UH    51    0    0 Eth1-1
127.1.0.0       0.0.0.0         255.255.0.0    U     0     0    0 veobc
127.1.2.0       0.0.0.0         255.255.255.0  U     0     0    0 veobc
[admin@guestshell ~]$
[admin@guestshell ~]$ ifconfig -a Eth1-1
Eth1-1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.102.242.131 netmask 255.255.255.240 broadcast 10.102.242.143
    ether 00:3a:9c:5a:00:67 txqueuelen 100 (Ethernet)
    RX packets 2044610 bytes 469523762 (447.7 MiB)
    RX errors 0 dropped 1614879 overruns 0 frame 0
    TX packets 556415 bytes 95505736 (91.0 MiB)
    TX errors 0 dropped 892 overruns 0 carrier 0 collisions 0
[admin@guestshell ~]$
[admin@guestshell ~]$ arp 10.102.242.129
Address          HWtype  HWaddress      Flags Mask      Iface
10.102.242.129  ether   00:1e:f7:be:70:c2  CM           Eth1-1
[admin@guestshell ~]$
```

Make sure the new route added in the default VRF context is synchronized to the global routing table. As shown in Example 7-12, a /16 route is added in the NX-OS, which has synchronized to the Guest Shell.

Example 7-12 *NX-OS and Guest Shell Synchronization*

```
N3K-C3548P#(config)# config t
Enter configuration commands, one per line. End with CNTL/Z.
N3K-C3548P(config)# ip route 192.168.0.0/16 10.102.242.129
N3K-C3548P(config)# end
N3K-C3548P# guestshell
[admin@guestshell ~]$
[admin@guestshell ~]$ chvrf default route -nv
Kernel IP routing table

```

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
0.0.0.0	10.102.242.129	0.0.0.0	UG	51	0	0	Eth1-1
10.1.1.0	0.0.0.0	255.255.255.0	U	0	0	0	Lo100
10.102.242.128	0.0.0.0	255.255.255.240	U	0	0	0	Eth1-1
10.102.242.129	0.0.0.0	255.255.255.255	UH	51	0	0	Eth1-1
127.1.0.0	0.0.0.0	255.255.0.0	U	0	0	0	veobc
127.1.2.0	0.0.0.0	255.255.255.0	U	0	0	0	veobc
192.168.0.0	10.102.242.129	255.255.0.0	UG	51	0	0	Eth1-1

```
[admin@guestshell ~]$
```

The NetBroker module synchronizes the ARP, routes, and other Layer 3 configuration to every kernel namespace available. Now you will switch to the Management namespace and verify the routes and ARP cache there (see Example 7-13).

Example 7-13 *Guest Shell Routing and ARP Tables—Management Namespace*

```
[admin@guestshell ~]$
[admin@guestshell ~]$ chvrf management
[admin@guestshell ~]$
[admin@guestshell ~]$ ifconfig -a
eth1: flags=4099<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.16.31.5 netmask 255.255.255.0 broadcast 172.16.31.255
    ether 00:3a:9c:5a:00:60 txqueuelen 1000 (Ethernet)
    RX packets 656019 bytes 48111417 (45.8 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 118874 bytes 31380645 (29.9 MiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
<snip>
[admin@guestshell ~]$ chvrf management route -vn
Kernel IP routing table
```

```

Destination      Gateway          Genmask          Flags Metric Ref  Use Iface
0.0.0.0          172.16.31.1     0.0.0.0          UG    51    0    0 eth1
<snip>
[admin@guestshell ~]$
[admin@guestshell ~]$ arp 172.16.31.1
Address          HWtype  HWaddress          Flags Mask          Iface
172.16.31.1     ether   00:1e:f7:a3:81:c6  CM                    eth1
[admin@guestshell ~]$

```

Installation and Verification of Applications

As you see in Example 7-14, the Guest Shell in Cisco Nexus 9000 Series devices supports Python version 2.7.5 in both interactive and noninteractive (script) modes.

The Python scripting capability in Nexus 9000 gives programmatic access to the device's command-line interface (CLI) to perform various tasks like Power On Auto Provisioning (POAP) and Embedded Event Manager (EEM).

Example 7-14 Python in Guest Shell

```

[admin@guestshell ~]$
[admin@guestshell ~]$ python
Python 2.7.5 (default, Jun 17 2014, 18:11:42)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-16)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello"
Hello
>>> quit()
[admin@guestshell ~]$

```

You will start with developing and running a Python application in the Guest Shell.

Custom Python Application

Python applications can be run from NX-OS using the `run guestshell python` command, or they can be natively run in the shell itself. As you see in Example 7-15, the Python application `hello.py` runs natively from NX-OS using the `run guestshell python` command and from the Guest Shell using the `python` command.

Example 7-15 Run Python Application in Guest Shell

```

N3K-C3548P#
N3K-C3548P# show file bootflash:hello.py
#!/usr/bin/env python

```

```

import sys

print "Hello, World!"
list = ['one', 'two', 'three']
for item in list:
    print item
N3K-C3548P#
N3K-C3548P# run guestshell python /bootflash/hello.py
Hello, World!
one
two
three
N3K-C3548P#
N3K-C3548P# guestshell
[admin@guestshell ~]$
[admin@guestshell ~]$ python /bootflash/hello.py
Hello, World!
one
two
three
[admin@guestshell ~]$ exit
N3K-C3548P#

```

Python API–Based Application

Cisco NX-OS has a built-in package providing API access to CLIs at the exec level as well as configuration commands, referred to as Python APIs. As you learned previously, Guest Shell also has access to Python APIs. As you see in Example 7-16, an NX-OS CLI **show clock** is accessed using the Python API available in the Guest Shell.

Example 7-16 *Python API–Based Application*

```

N3K-C3548P#
N3K-C3548P# guestshell
[admin@guestshell ~]$
[admin@guestshell ~]$ python
Python 2.7.5 (default, Jun 17 2014, 18:11:42)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-16)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from cli import *
>>> cli('show clock')
'02:24:50.130 UTC Sun Sep 01 2019\nTime source is NTP\n'
>>> exit()
[admin@guestshell ~]$

```

Example 7-17 shows a sample custom Python application that leverages Python APIs. In this example, `cli` returns the raw format of the CLI output, including control and special characters. `clid` returns a dictionary of attribute names and values for the given CLI commands, which makes it easier to process the data programmatically and automate.

Example 7-17 *Python API-Based Application: JSON*

```
[admin@guestshell ~]$ more PY-API2.py

#!/usr/bin/python
from cli import *
import json

print("STANDARD CLI OUTPUT ...")
print (cli('show interface eth1/1 brief'))

print("JSON FORMAT CLI OUTPUT ...")
print (clid('show interface eth1/1 brief'))

[admin@guestshell ~]$
[admin@guestshell ~]$
[admin@guestshell ~]$ python PY-API2.py
STANDARD CLI OUTPUT ...
-----
Ethernet  VLAN    Type Mode   Status Reason   Speed   Port
Interface                                     Ch #
-----
Eth1/1    --      eth  routed up     none    1000(D)  --

JSON FORMAT CLI OUTPUT ...
{"TABLE_interface": {"ROW_interface": {"interface": "Ethernet1/1", "vlan": "--",
  "type": "eth", "portmode": "routed", "state": "up", "state_rsn_desc": "none",
  "speed": "1000", "ratemode": "D"}}}}
[admin@guestshell ~]$
```

To learn more about Python APIs and the Software Development Kit (SDK) supported in Nexus 9000 platforms, refer to the *Cisco Nexus 9000 Series SDK User Guide* provided in the “References” section.

The `dohost` command shown in Example 7-18 is a Python wrapper script using NX-API functions. Make sure to have the NX-API feature enabled to leverage this capability. Using `dohost` capability, application developers can perform `show` commands as well as configuration commands.

Example 7-18 *Run NX-OS CLIs in Guest Shell with dohost*

```
[admin@guestshell ~]$
[admin@guestshell ~]$ dohost "show clock"
02:23:41.492 UTC Sun Sep 01 2019
Time source is NTP
```

As you learned in the previous section, the Guest Shell with CentOS 7 also can install software packages using Yum utility. The Guest Shell is prepopulated with many of the common tools that would naturally be expected on a networking device, including net-tools, iproute, tcpdump, OpenSSH, and the PIP for installing additional Python packages. As you have just learned, Python 2.7.5 is included by default.

Leveraging high-end capabilities and features in Guest Shell, it is easy to integrate it into your day-to-day automation workflow. With the support of device-level API integration and support for scripting with languages like Python, Ruby, and so on, it is easier now to do on-box prototyping of applications or scripts. Guest Shell has its user space and resources isolated from the host and other containers and any faults/failures seen in those container spaces. All the capabilities make Guest Shell a powerful environment to develop and host applications.

Bash

In addition to the NX-OS CLI, Cisco Nexus 9000 Series devices support access to Bash. Bash interprets commands that you enter or commands that are read from a shell script. It enables access to the underlying Linux kernel on the device and to manage the system.

As you learned in the previous sections, Bash is supported in Nexus 3000 and 9000 switching platforms, but it is disabled by default.

Enabling Bash

In the supported platforms, under configuration mode, the **feature bash-shell** command enables this feature with no special license required. Use the **show bash-shell** command to learn the current state of the feature, as shown in Example 7-19.

Example 7-19 *Check Status and Enable Bash*

```
N9K-C93180YC# show bash-shell
Bash shell is disabled
N9K-C93180YC#
N9K-C93180YC# conf t
Enter configuration commands, one per line. End with CNTL/Z.
N9K-C93180YC(config)# feature bash-shell
N9K-C93180YC(config)# end
```

```

N9K-C93180YC#
N9K-C93180YC# show bash-shell
Bash shell is enabled
N9K-C93180YC#

```

Accessing Bash from NX-OS

In Cisco NX-OS, Bash is accessible for users whose role is set to network-admin or dev-ops; through Bash, a user can change system settings or parameters that could impact devices' operation and stability.

You can execute Bash commands with the **run bash** command, as shown in Example 7-20.

Example 7-20 *Run Bash Commands from NX-OS*

```

N9K-C93180YC#
N9K-C93180YC# run bash pwd
/bootflash/home/admin
N9K-C93180YC#
N9K-C93180YC# run bash ls
N9K-C93180YC# run bash uname -r
4.1.21-WR8.0.0.25-standard
N9K-C93180YC#
N9K-C93180YC# run bash more /proc/version
Linux version 4.1.21-WR8.0.0.25-standard (divvenka@ins-ucs-bld8) (gcc version 4.6.3
(Wind River Linux Sourcery CodeBench 4.6-60) ) #1 SMP Sun Nov 4 19:44:18 PST 2018
N9K-C93180YC#
N9K-C93180YC#

```

The **run bash** command loads Bash and begins at the home directory for the user. Example 7-21 shows how to load and run Bash as an admin user.

Example 7-21 *Access Bash Through Console*

```

N9K-C93180YC#
N9K-C93180YC# run bash
bash-4.3$
bash-4.3$ pwd
/bootflash/home/admin
bash-4.3$
bash-4.3$ whoami
admin
bash-4.3$
bash-4.3$ id

```

```
uid=2002(admin) gid=503(network-admin) groups=503(network-admin),504(network-
operator)
bash-4.3$
bash-4.3$ more /proc/version
Linux version 4.1.21-WR8.0.0.25-standard (divvenka@ins-ucs-bld8) (gcc version 4.
6.3 (Wind River Linux Sourcery CodeBench 4.6-60) ) #1 SMP Sun Nov 4 19:44:18 PST
2018
bash-4.3$
```

For users without network-admin or dev-ops level privileges, the **run bash** command will not be parsed, and when executed, the system will report that permission has been denied. As you see in Example 7-22, the **testuser** with the privilege level not set to network-admin or dev-ops has its permission to execute the **run bash** command denied.

Example 7-22 Access Bash Privileges

```
User Access Verification
N9K-C93180YC login: testuser
Password:
Cisco Nexus Operating System (NX-OS) Software
TAC support: http://www.cisco.com/tac
Copyright (C) 2002-2018, Cisco and/or its affiliates.
All rights reserved.
<snip>
N9K-C93180YC# run bash
% Permission denied for the role
N9K-C93180YC#
```

Accessing Bash via SSH

Before accessing Bash via SSH, make sure the SSH service is enabled (see Example 7-23).

Example 7-23 Access Bash Privileges

```
bash-4.3$ service /etc/init.d/sshd status
openssh-daemon (pid 14190) is running...
bash-4.3$
bash-4.3$ ps -ef | grep sshd
UID          PID  PPID  C  STIME TTY          TIME CMD
admin        5619  5584  0  01:26 ttyS0        00:00:00 grep sshd
root         14190    1   0  Sep12 ?           00:00:00 /usr/sbin/sshd
bash-4.3$
bash-4.3$ ps --pid 1
  PID TTY          TIME CMD
   1 ?           00:00:28 init
bash-4.3$
```

An NX-OS admin user can configure a user with privileges to directly log in to the Bash. Example 7-24 demonstrates user **bashuser** with a default shelltype access.

Example 7-24 *Access Bash Privileges: shelltype*

```
N9K-C93180YC#
N9K-C93180YC# conf t
Enter configuration commands, one per line. End with CNTL/Z.
N9K-C93180YC(config)#
N9K-C93180YC(config)# username bashuser password 0 Cisco!123
N9K-C93180YC(config)# username bashuser shelltype bash
N9K-C93180YC(config)# end
N9K-C93180YC#
```

Log in to Bash directly from an external device with username **bashuser**, as shown in Example 7-25.

Example 7-25 *Access Bash—Shelltype User*

```
Ubuntu-Server$ ssh -l bashuser 172.16.28.5
User Access Verification
Password:
-bash-4.3$
-bash-4.3$ pwd
/var/home/bashuser
-bash-4.3$
-bash-4.3$ id
uid=2003(bashuser) gid=504(network-operator) groups=504(network-operator)
-bash-4.3$
-bash-4.3$ whoami
bashuser
-bash-4.3$
-bash-4.3$ exit
logout
Connection to 10.102.242.131 closed.
Ubuntu-Server$
```

Following are the guidelines for elevating the privileges of an existing user.

- Bash must be enabled before elevating user privileges.
- Only an admin user can escalate privileges of a user to root.
- Escalation to root is password protected.

If you SSH to the switch using the **root** username through a nonmanagement interface, it will default to Linux Bash shell-type access for the root user. If a user has established

an SSH connection directly to Bash and needs to access NX-OS, use `vsh` commands, as shown in Example 7-26.

Example 7-26 *Access NX-OS from Bash*

```

bash-4.3$
bash-4.3$ vsh -c "show clock"
21:17:24:136 UTC Fri Sep 13 2019
Time source is NTP
bash-4.3$
bash-4.3$ su - root
Password:
root@N9K-C93180YC#
root@N9K-C93180YC# id
uid=0(root) gid=0(root) groups=0(root)
root@N9K-C93180YC# whoami
root
root@N9K-C93180YC#
root@N9K-C93180YC# vsh
Cisco Nexus Operating System (NX-OS) Software
TAC support: http://www.cisco.com/tac
Copyright (C) 2002-2018, Cisco and/or its affiliates.
All rights reserved.
<snip>
root@N9K-C93180YC#
root@N9K-C93180YC# show clock
21:18:53.903 UTC Fri Sep 13 2019
Time source is NTP
root@N9K-C93180YC#

```

Based on what you have learned this section, Bash interprets the instructions and commands that a user or application provides and executes. With direct access to the underlying infrastructure, file systems, and network interfaces, it enables developers to build and host applications to monitor and manage the devices. However, users should exercise extreme caution when accessing, configuring, or making changes to the underlying infrastructure because doing so could affect the host system's operation and performance. Remember that Bash directly accesses the Wind River Linux (WRL) on which NX-OS is running in a user space, and unlike Guest Shell or OAC, it is not isolated from the host system.

Docker Containers

Docker provides a way to securely run applications in an isolated environment, with all dependencies and libraries packaged. If you want to know more about Docker, its usage, and functionalities, refer to the *Docker Documentation* page provided in the "References" section.

Beginning with Release 9.2(1), support has been included for using Docker within the Cisco NX-OS switch. The version of Docker that is included on the switch is 1.13.1. By default, the Docker service or daemon is not enabled. You must start it manually or set it up to automatically restart when the switch boots up.

Even though the scope of this book does not intend to cover Docker in detail, it is good to take a quick look at the key components in the Docker architecture and their functions, as illustrated in Figure 7-6.

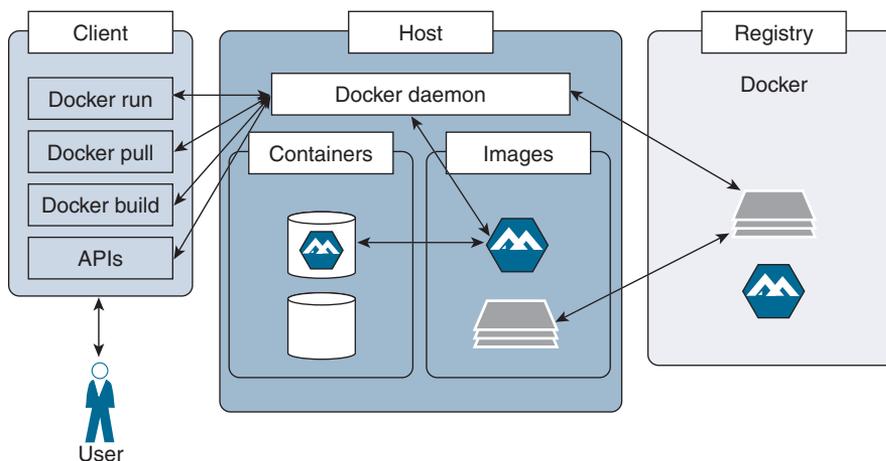


Figure 7-6 *Docker Architecture*

Docker Client

The Docker client enables end users to interact with the Docker host and the daemons running on it. The Docker client can be on a dedicated device or can reside on the same device as a host. A Docker client can communicate with multiple daemons running on multiple host devices. The Docker client provides a CLI and REST APIs that allow users to issue build, run, and stop application commands to a Docker daemon. The main purpose of the Docker client is to provide a means to direct pulling images from a registry and having them run on a Docker host.

Docker Host

The Docker host provides an environment dedicated to executing and running applications. The key component is a Docker daemon that interacts with the client as well as the registry and with containers, images, the network, and storage. This daemon is responsible for all container-related activities and carrying out the tasks received via CLIs or APIs. The Docker daemon pulls the requested image and builds containers as requested by the client, following the instructions provided in a build file.

Images

Images are read-only templates providing instructions to create a Docker container. The images contain metadata that describe the container's capabilities and needs. The necessary Docker images can be pulled from the Docker Hub or a local registry. Users can create their own and customized images by adding elements to extend the capabilities, using Dockerfile.

Containers

As has been discussed in previous chapters, containers are self-contained environments in which you run applications. The container is defined by the image and any additional configuration parameters provided during its instantiation. These configuration parameters are used to identify the file systems and partitions to mount, to set specific network mode, and so on.

Now you will learn how to enable and use Docker in the context of the Cisco Nexus switch environment.

Bash is a prerequisite to enable and activate Docker. Example 7-27 provides the detailed procedure to activate Docker. Before activating Docker, follow these steps.

1. Enable Bash.
2. Configure the domain name and name servers appropriately for the network.
3. If the switch is in a network that uses an HTTP proxy server, set up the `http_proxy` and `https_proxy` environment variables in the `/etc/sysconfig/docker` file.

Example 7-27 *Enable Bash to Activate Docker Service*

```
N9K-C93180YC# conf t
N9K-C93180YC(config)# feature bash-shell
N9K-C93180YC(config)# vrf context management
N9K-C93180YC(config-vrf)# ip domain-name cisco.com
N9K-C93180YC(config-vrf)# ip name-server 208.67.222.222
N9K-C93180YC(config-vrf)# ip name-server 208.67.220.220
N9K-C93180YC(config-vrf)# end
N9K-C93180YC# run bash
bash-4.3$
bash-4.3$ cat /etc/resolv.conf
domain cisco.com
nameserver 208.67.222.222
nameserver 208.67.220.220
bash-4.3$
bash-4.3$ cat /etc/sysconfig/docker | grep http
export http_proxy=http://192.168.21.150:8080
export https_proxy=http://192.168.21.150:8080
bash-4.3$
```

Starting Docker Daemon

Please be aware that when the Docker daemon is started for the first time, 2 GB of storage space is carved out for a file called **dockerpart** in the bootflash filesystem. This file will be mounted as `/var/lib/docker`. If needed, the default size of this space reservation can be changed by editing `/etc/sysconfig/docker` before you start the Docker daemon for the first time.

Start the Docker daemon by following Example 7-28.

Example 7-28 *Enable Docker Service*

```
bash-4.3$
bash-4.3$ service docker start
bash-4.3$
bash-4.3$ service docker status
dockerd (pid 5334) is running...
bash-4.3$
bash-4.3$ ps -ef | grep docker
UID    PID    PPID    C    STIME TTY      TIME    CMD
root   16532     1     0   03:15 ttyS0    00:00:00 /usr/bin/dockerd --debug=true
root   16548  16532     0   03:15 ?        00:00:00 docker-containerd -l unix:///var
admin  16949  12789     0   03:18 ttyS0    00:00:00 grep docker
bash-4.3$
bash-4.3$
```

Instantiating a Docker Container with Alpine Image

As you can see in Example 7-29, the host device has various Docker images, including Alpine, Ubuntu, and nginx. Alpine Linux is a lightweight Linux distribution based on musl libc and Busybox, and it is security-oriented. Musl (read as, “muscle”) libc is a standard library of Linux-based devices focused on standards-conformance and safety. Busybox brings many UNIX/Linux utilities together into a single and small executable; because it is modular, it is easy to customize and integrate it into embedded systems. For more information, see the references provided for Alpine Linux, musl libc, and Busybox, in the “References” section at the end of this chapter.

Example 7-29 shows instantiating an Alpine Linux Docker container on the switch, which is, by default, launched in the host network mode. The Docker containers instantiated in the **bridged** networking mode have external network connectivity but do not necessarily care about the visibility into or access to ports in the host. Note that the containers operating in **bridged** networking mode are far more secure than the ones operating in **host** networking mode.

Example 7-29 *Container with Alpine Image*

```

bash-4.3$
bash-4.3$ docker images
REPOSITORY          TAG                IMAGE ID           CREATED            SIZE
docker              dind              12adad4e12e2     3 months ago     183 MB
ubuntu              latest            d131e0fa2585     4 months ago     102 MB
nginx                latest            27a188018e18     5 months ago     109 MB
alpine               latest            cdf98d1859c1     5 months ago     5.53 MB
centos               latest            9f38484d220f     6 months ago     202 MB
alpine               3.2               98f5f2d17bd1     7 months ago     5.27 MB
hello-world         latest            fce289e99eb9     8 months ago     1.84 kB
bash-4.3$
bash-4.3$
bash-4.3$ docker run --name=myalpine -v /var/run/netns:/var/run/netns:ro,rslave
--rm --network host --cap-add SYS_ADMIN -it alpine
/ #
/ # whoami
root
/ # id
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy),20(dialout),26(tape),27(video)
/ #
/ # ip route
default via 10.102.242.129 dev Eth1-1 metric 51 onlink
10.1.1.0/24 dev Lo100 scope link
10.102.242.128/28 dev Eth1-1 scope link
10.102.242.129 dev Eth1-1 scope link metric 51
127.1.0.0/16 dev veobc scope link src 127.1.1.1
127.1.2.0/24 dev veobc scope link src 127.1.2.1
172.17.0.0/16 dev docker0 scope link src 172.17.0.1
172.18.0.0/16 dev br-b96ec30eb010 scope link src 172.18.0.1
172.16.0.0/16 via 10.102.242.129 dev Eth1-1 metric 51 onlink
/ #
/ # ifconfig Eth1-1
Eth1-1    Link encap:Ethernet  Hwaddr 00:3A:9C:5A:00:67
          inet addr:10.102.242.131 Bcast:10.102.242.143 Mask:255.255.255.240
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2873124 errors:0 dropped:2299051 overruns:0 frame:0
          TX packets:797153 errors:0 dropped:1230 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:622065894 (593.2 MiB)  TX bytes:135952384 (129.6 MiB)
/ #

```

Figure 7-7 illustrates a Docker container running an Alpine image that was instantiated from Bash by the commands provided in Example 7-29.

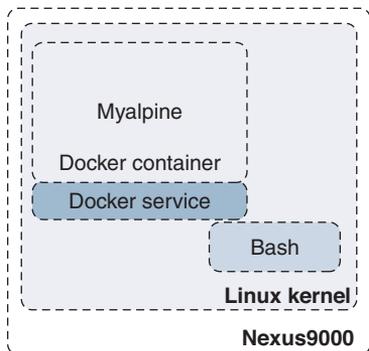


Figure 7-7 *Alpine Docker Container*

The `-rm` option used to launch the Docker container in Example 7-29 removes it automatically when the user exits the container with the `exit` command. Press `Ctrl+Q` to detach from the container without deinstantiating it and get back to Bash. Use the `docker attach <container-id>` command to reattach to the container that is still up and running, as shown in Example 7-30.

Example 7-30 *Docker Processes—Attach to Container*

```
bash-4.3$
bash-4.3$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
6469af028115  alpine   "/bin/sh" 3 minutes ago  Up 3 minutes           myalpine
bash-4.3$
bash-4.3$ docker attach 6469af028115
/ #
/ #
```

If you want to mount a specific file system or partitions, use the `-v` option, as shown in Example 7-31, when you launch the container. The Bootflash file system will be mounted into and accessible only from the `myalpine1` container; it will not be available from `myalpine`, which was instantiated without mounting the Bootflash file system.

Example 7-31 *Docker Container—File System Mount*

```
bash-4.3$
bash-4.3$ docker run --name=myalpine1 -v /var/run/netns:/var/run/netns:ro,rslave -v
/bootflash:/bootflash --rm --network host --cap-add SYS_ADMIN -it alpine
/ #
```

```

/ # ls
bin      etc      media   proc    sbin    tmp
bootflash home    mnt     root    srv     usr
dev      lib     opt     run     sys     var
/ # / # ifconfig
Eth1-1   Link encap:Ethernet  Hwaddr 00:3A:9C:5A:00:67
        inet addr:10.102.242.131 Bcast:10.102.242.143 Mask:255.255.255.240
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:2848104 errors:0 dropped:2282704 overruns:0 frame:0
        TX packets:786971 errors:0 dropped:1209 overruns:0 carrier:0
        collisions:0 txqueuelen:100
        RX bytes:618092996 (589.4 MiB)  TX bytes:134371507 (128.1 MiB)

Eth1-10  Link encap:Ethernet  Hwaddr 00:3A:9C:5A:00:67
        UP BROADCAST MULTICAST  MTU:1500  Metric:1
<snip>

```

The Alpine Docker containers instantiated in the past few examples were done in the default host namespace. To instantiate a Docker container in a specific network namespace, use the `docker run` command with the `-network <namespace>` option.

Managing Docker Container

Beyond instantiating and activating containers with applications installed, you need to know how to manage the containers. Container management becomes critical when containers are deployed at scale. This section discusses managing containers deployed in the Nexus switches, and associated techniques.

Container Persistence Through Switchover

To have Docker container persisting through the manual supervisor engine switchover, make sure to copy the `dockerpart` file from the active supervisor engine's bootflash to the standby supervisor engine's bootflash before the switchover of supervisor engines in applicable platforms like Nexus 9500. Be aware that the Docker containers will not be running continuously and will be disrupted during the switchover.

You will start an Alpine container and configure it to always restart unless it is explicitly stopped or the Docker service is restarted. Please note that this command uses the `-restart` option instead of the `-rm` option, which restarts the container right after the user exits. See Example 7-32.

Example 7-32 Docker Container—Persistent Restart

```

bash-4.3$
bash-4.3$ docker run -dit --name=myalpine2 --restart unless-stopped --network host
--cap-add SYS_ADMIN -it alpine
da28182a03c4032f263789ec997eea314130a95e6e6e6a0574e49dfc5f2776

```

```

bash-4.3$
bash-4.3$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
0355f5ba1fd6  alpine   "/bin/sh" 18 minutes ago Up 5 minutes      myalpine2
bash-4.3$
bash-4.3$ docker attach 0355f5ba1fd6
/#
/# exit
bash-4.3$
bash-4.3$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
0355f5ba1fd6  alpine   "/bin/sh" 19 minutes ago Up 2 seconds      myalpine2
bash-4.3$

```

With the previous commands, you have made the Alpine Linux container restart. As shown in Example 7-33, use the **chkconfig** utility to make the service persistent, before the supervisor engine switchover. Then copy the dockerpart file created in the active supervisor engine to standby.

Example 7-33 Docker Container—Restart on Supervisor Engine Failover

```

bash-4.3$
bash-4.3$ chkconfig | grep docker
bash-4.3$
bash-4.3$ chkconfig --add docker
bash-4.3$
bash-4.3$ chkconfig | grep docker
docker          0:off  1:off  2:on   3:on   4:on   5:on   6:off
bash-4.3$
bash-4.3$ service docker stop
Stopping docker: dockerd shutdown
bash-4.3$
bash-4.3$ cp /bootflash/dockerpart /bootflash_sup-remote/
bash-4.3$
bash-4.3$ service docker start
bash-4.3$

```

Stopping the Docker Container and Service

If a specific container needs to be stopped, use the **docker stop** command, as shown in Example 7-34. To learn more Docker command options, use the **docker -help** and **docker run -help** commands.

When a specific container is stopped, all the applications, along with their packages and libraries, will cease to function, and any file system mounted will be unmounted.

Example 7-34 *Stopping the Docker Container*

```

bash-4.3$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS          NAMES
0355f5ba1fd6  alpine   "/bin/sh"               36 minutes ago  Up 13 minutes  myalpine2
bash-4.3$
bash-4.3$ docker stop 0355f5ba1fd6
0355f5ba1fd6
bash-4.3$

```

If a Docker service needs to be stopped altogether, follow the procedure as given in Example 7-35. As you have learned, if a Docker service is not up and running, containers will cease to exist in Nexus switches. Make sure to delete the dockerpart file from the active supervisor engine's bootflash as well as the standby's bootflash in applicable deployment scenarios.

Example 7-35 *Stopping the Docker Service*

```

bash-4.3$
bash-4.3$ service docker stop
Stopping dockerd: dockerd shutdown
bash-4.3$
bash-4.3$ service docker status
dockerd is stopped
bash-4.3$ exit
N9K-C93180YC#
N9K-C93180YC# delete bootflash:dockerpart
Do you want to delete "/dockerpart" ? (yes/no/abort) y
N9K-C93180YC#

```

Orchestrating Docker Containers Using Kubernetes

Kubernetes is an open-source platform for automating, deploying, scaling, and operating containers. Kubernetes was first created by Google and then donated to Cloud Native Compute Foundation (open source). Since Kubernetes became open source, there have been several projects to increase its scope and improve it to enable networking, storage, and more, which allows users to focus on developing and testing applications rather than spending resources to gain expertise in and maintain container infrastructure.

Kubernetes Architecture

Following is a brief discussion on the Kubernetes architecture, which will help you follow the procedures and examples provided later.

In a Kubernetes (or K8s) cluster functionally, there are two major blocks—Master and Node—as illustrated in Figure 7-8.

Master components provide the cluster’s control plane. Master components make global decisions about the cluster (for example, scheduling) and detect and respond to cluster events (for example, starting up a new pod). Master components are kube-apiserver, etcd, kube-scheduler, kube-controller-manager, and cloud-controller-manager. Master components can be run on any machine in the cluster, and it is highly recommended that you have all master components running in the same machine, where no containers are instantiated.

Node components run on every host or a virtual machine, maintaining pods deployed and providing the Kubernetes runtime environment. Node components are kubelet, kube-proxy, and container runtime.

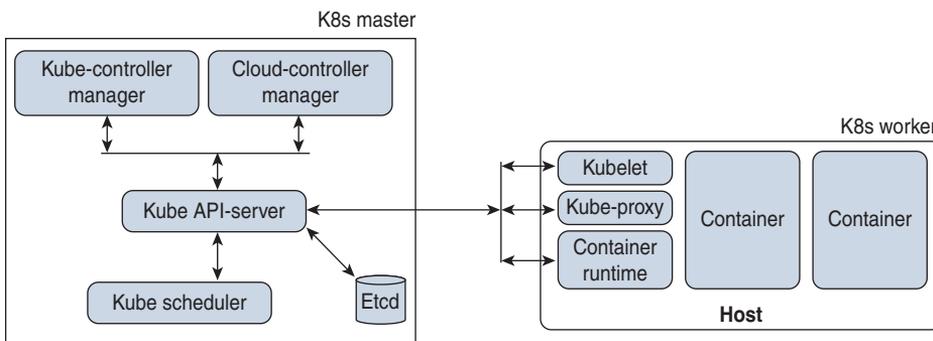


Figure 7-8 *Kubernetes Architecture*

The Cloud controller manager is a daemon that has the cloud-specific control loops. The Kubernetes controller manager is a daemon that has the core control loops. In K8s, a controller is a control loop that monitors the state of the cluster through the API server and makes necessary changes to move the current state toward the desired state. Examples of controllers that ship with Kubernetes are the replication controller, endpoints controller, namespace controller, and service accounts controller.

You will take a quick look at the common terminologies used in the Docker containers and Kubernetes world.

Pod

A pod is a group of containers sharing resources such as volumes, file systems, storage, and networks. It also is a specification on how these containers are run and operated. In a simple view, a pod is synonymous to an application-centric logical host, which contains one or more tightly coupled containers. In a given pod, the containers share an IP address and Layer 4 port space and can communicate with each other using standard interprocess communication.

Controllers

Kubernetes contains many higher-level abstractions called controllers. Controllers build upon the basic objects and provide additional functionality and convenience features, such as ReplicaSet, StatefulSet, and DaemonSet.

The objective of a ReplicaSet is to maintain a set of replica pods running at any given time, guaranteeing the availability of a specified number of identical pods.

StatefulSet is the workload API object used to manage stateful applications. It manages the deployment and scaling of a set of pods **and guarantees the ordering and uniqueness** of these pods.

A DaemonSet is an object that ensures that all or some of the nodes run a copy of a pod. As a cluster expands by adding more nodes, DaemonSet makes sure that pods are added to the new added nodes. When nodes are removed from the cluster, those pods are removed, and the garbage is collected.

If you need more information on Kubernetes, please see the Kubernetes page at <https://kubernetes.io/>.

Building Kubernetes Master

You are going to build a K8s Master in an Ubuntu server, as shown in Example 7-36.

A K8s Master can be run natively in a Linux environment such as Ubuntu. But for convenience, you will run the K8s Master as a Docker container. The command provided in the example enables the Docker service to prepare the Ubuntu server for running Kubernetes Master components. Note that the following example uses Kubernetes version 1.2.2.

Example 7-36 *Building K8s Master—Docker Service*

```
root@Ubuntu-Server1$
root@Ubuntu-Server1$ service docker start
root@Ubuntu-Server1$
root@Ubuntu-Server1$ service docker status
dockerd (pid 17362) is running...
root@Ubuntu-Server1$
```

etcd is a highly available database of the K8s Master, which has all cluster data in a key-value pair format. As shown in Example 7-37, the **docker run** command starts the etcd component. The IP address and TCP port it is listening to are 10.0.0.6 and 4001, respectively.

Example 7-37 *Building K8s Master—etcd*

```
root@Ubuntu-Server1$ docker run -d \
  --net=host \
  gcr.io/google_containers/etcd:2.2.1 \
  /usr/local/bin/etcd --listen-client-urls=http://10.0.0.6:4001 \
  --advertise-client-urls=http://10.0.0.6:4001 --data-dir=/var/etcd/data
```

As you notice in Example 7-38, the K8s Master components API server is started, and it is listening to the same IP address and TCP port as etcd.

Example 7-38 *Building K8s Master—API Server*

```

root@Ubuntu-Server1$ docker run -d --name=api \
  --net=host --pid=host --privileged=true \
  gcr.io/google_containers/hyperkrs/hyperkubeube:v1.2.2 \
  /hyperkube apiserver --insecure-bind-address=10.0.0.6 \
  --allow-privileged=true \
  --service-cluster-ip-range=172.16.1.0/24 \
  --etcd_servers=http://10.0.0.6:4001 --v=2

```

The next step is to start the kubelet of the K8s Master components. The kubelet is listening to the same IP address as the `etcd` or the API server, but the TCP port is 8080. Please follow the steps provided in Example 7-39 to start the kubelet.

Example 7-39 *Building K8s Master—Kubelet*

```

root@Ubuntu-Server1$ docker run -d --name=kubs \
  --volume=/:/rootfs:ro --volume=/sys:/sys:ro --volume=/dev:/dev \
  --volume=/var/lib/docker:/var/lib/docker:rw \
  --volume=/var/lib/kubelet:/var/lib/kubelet:rw \
  --volume=/var/run:/var/run:rw --net=host --pid=host \
  --privileged=true \
  gcr.io/google_containers/hyperkube:v1.2.2 \
  /hyperkube kubelet --allow-privileged=true \
  --hostname-override="10.0.0.6" \
  --address="10.0.0.6 --api-servers=http://10.0.0.6:8080 \
  --cluster_dns=10.0.0.10 \
  --cluster_domain=cluster.local --config=/etc/kubernetes/manifests-multi

```

The last step you need to do in the Master is to enable kube-proxy. It is a network proxy that runs on each node in your cluster, and it maintains the network rules on nodes. These network rules allow network communication to your pods from network sessions inside or outside your cluster. kube-proxy uses the operating system packet filtering layer if it is available. Enable kube-proxy as shown in Example 7-40.

Example 7-40 *Building K8s Master—Kube Proxy*

```

root@Ubuntu-Server1$ docker run -d --name=proxy --net=host --privileged gcr.io/
google_containers/hyperkube:v1.2.2/hyperkube proxy --master=http://10.0.0.6:8080
--v=2

```

Figure 7-9 illustrates the K8s Master running in an Ubuntu server and various components in the K8s Master.

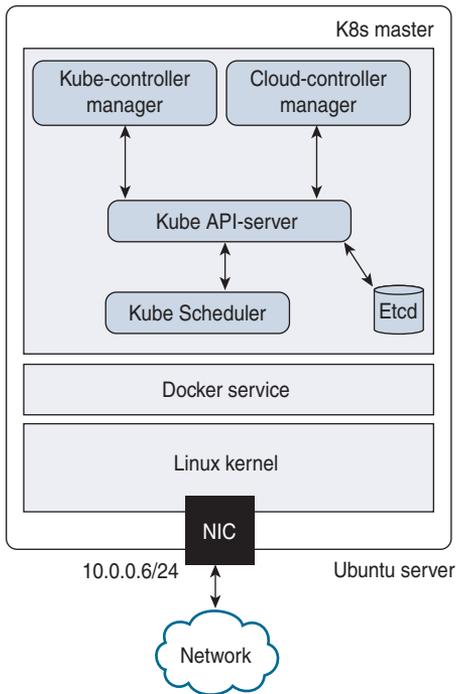


Figure 7-9 *Kubernetes Master—Ubuntu Server*

Now that you have a K8s Master service running, register Nexus 9000 as a node to the K8s Master. As you see in Example 7-41, the `docker run` commands register to the Master and the socket to which the kube-apiserver and other Master components are listening.

Example 7-41 *Register Nexus Switch as K8s Node to Master*

```
N9K-C93180YC# run bash
bash-4.3$
bash-4.3$ docker run -d --name=kubs --net=host --pid=host --privileged=true
--volume=/:/rootfs:ro --volume=/sys:/sys:ro --volume=/dev:/dev --volume=/var/
lib/docker/:/var/lib/docker:rw --volume=/var/lib/kubelet/:/var/lib/kubelet:rw
--volume=/var/run:/var/run:rw \ gcr.io/google_containers/hyperkube:v1.2.2/
hyperkube kubelet --allow-privileged=true --containerized --enable-server
--cluster_dns=10.0.0.10 \--cluster_domain=cluster.local --config=/etc/
kubernetes/manifests-multi \--hostname-override="10.0.0.6" --address=0.0.0.0
--api-servers=http://10.0.0.6:4001
bash-4.3$
bash-4.3$ docker run --name=proxy \--net=host --privileged=true gcr.io/google_
containers/hyperkube:v1.2.2 /hyperkube proxy --master=http://10.0.0.6:4001 --v=2
bash-4.3$
```

Once the Nexus 9000 successfully registers as a K8s Node to the Master, it should begin to communicate with the Master. Figure 7-10 shows a Kubernetes Cluster, with an Ubuntu server acting as a K8s Master and a Nexus 9000 acting as a K8s Node.

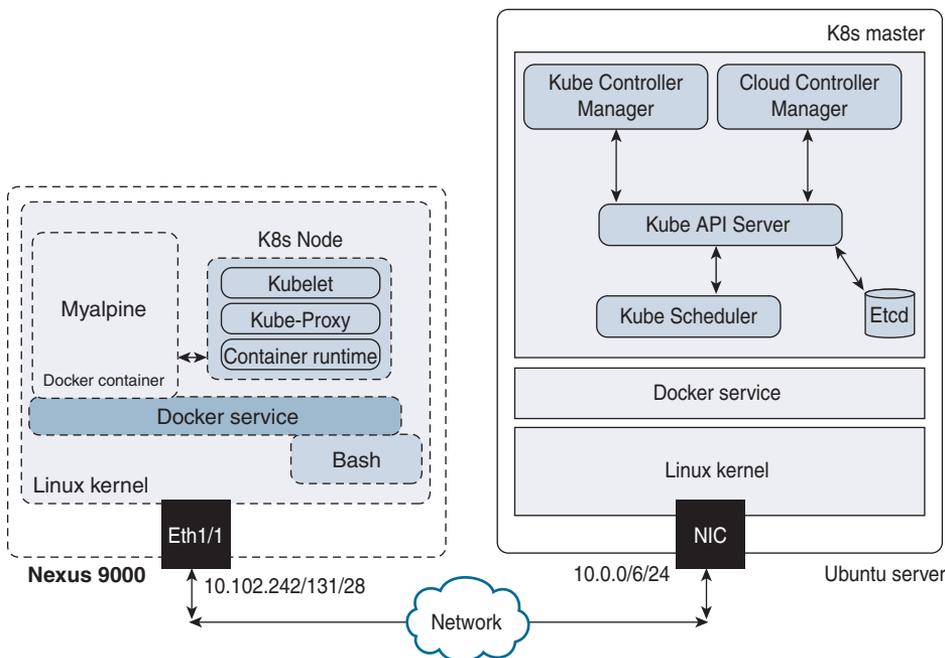


Figure 7-10 *Kubernetes Cluster*

The certificate exchange must happen between the Master and Node to establish a secure connection between them, so all the data and control message communication happens securely.

Orchestrating Docker Containers in a Node from the K8s Master

Now you will look into orchestration of Docker containers in a pod from the K8s Master and how you can manage them through their lifecycles. Kubectl is a critical component in managing and orchestrating containers.

Kubectl is a set of CLI commands to manage Kubernetes clusters. It can deploy applications and inspect and manage cluster resources, among other tasks.

Download and install kubectl packages in an Ubuntu server in which you have already instantiated the K8s Master. Example 7-42 shows using the `curl` command to download a specific version—in this case, it is v1.15.2. If you want to download a different version, replace v1.15.2 with the preferred version.

Example 7-42 *Install Kubectl in K8s Master*

```

root@Ubuntu-Server1$ curl -o ~/.bin/kubectl http://storage.googleapis.com/
kubernetes-release/release/v1.15.2/bin/linux/amd64/kubectl
root@Ubuntu-Server1$

```

Change the permissions to make the binary executable, and move it into the executable path, as shown in Example 7-43.

Example 7-43 *Make Kubectl Executable*

```

root@Ubuntu-Server1$ chmod u+x ./kubectl
root@Ubuntu-Server1$ mv ./kubectl /usr/local/bin/kubectl

```

By default, kubectl configuration is located in the `~/.kube/config` file. For kubectl to discover and access a Kubernetes cluster, it looks for the `kubeconfig` file in the `~/.kube` directory, which is created automatically when your cluster is created.

This `kubeconfig` file organizes information about clusters, users, namespaces, and authentication mechanisms. The kubectl command uses `kubeconfig` files to find the information it needs to choose a cluster and communicate with the API server of a cluster. If required, you can use the `-kubeconfig` flag to specify other `kubeconfig` files.

To learn how to install kubectl on different operating systems like Microsoft Windows or Apple macOS, please refer to the Install and Setup Kubectl Guide provided in the References section. Table 7-6 shows the kubectl syntax for common operations with examples, such as **apply**, **get**, **describe**, and **delete**. Note that the filenames used in the following table are for illustrative purposes only.

Table 7-6 *Kubectl Operations and Commands*

Operations	Commands
Create a service using the definition in the <code>example-service.yaml</code> file	<code>kubectl apply -f example-service.yaml</code>
Create a replication controller using the definition in a YAML file	<code>kubectl apply -f example-controller.yaml</code>
Create the objects that are defined in any <code>.yaml</code> , <code>.yml</code> , or <code>.json</code> files in a specific directory	<code>kubectl apply -f <directory></code>
List all pods in plain-text output format	<code>kubectl get pods <pod-name></code>
Get a list of all pods in plain-text output format and include additional information (node name, etc.)	<code>kubectl get pods -o wide</code>
Get a list of pods sorted by name	<code>kubectl get pods --sort-by=.metadata.name</code>

Operations	Commands
Get a list of all pods running on node by name	<code>kubectrl get pods --field-selector=spec.nodeName=<node-name></code>
Display the details of the node with node name	<code>kubectrl describe nodes <node-name></code>
Display the details of the pod with pod name	<code>kubectrl describe pods/<pod-name></code>
Delete a pod using the label	<code>Kubectrl delete pods -l name=<label></code>
Delete a pod using the type and name specified in a YAML file	<code>kubectrl delete -f pod.yaml</code>
Delete all pods—initialized as well as uninitialized ones	<code>kubectrl delete pods --all</code>

For details about each operation command, including all the supported flags and subcommands, see the *Kubectrl Overview* document provided in the “References” section.

Now that you have learned about *kubectrl*, you will see how to use it to manage clusters and nodes. In this case, the Kubernetes clusters have the Ubuntu server as K8s Master, the Nexus 9000 as Node, and an application named Alpine deployed. Example 7-44 shows *kubectrl* commands to get the nodes, deployment, and pods from the K8s Master. The command results indicate that an application is running as container *myalpine* in the K8s pods.

Example 7-44 *Use Kubectrl to Get Nodes, Deployments, and Pods*

```

root@Ubuntu-Server1$
root@Ubuntu-Server1$ kubectrl get nodes
NAME                STATUS    ROLES    AGE    VERSION
Ubuntu-Server1     Ready    master   11m    v1.2.2
N9K-C93180YC       Ready    <none>   18m    v1.2.2
root@Ubuntu-Server1$
root@Ubuntu-Server1$ kubectrl get deployments
NAME    READY    UP-TO-DATE    AVAILABLE    AGE
alpine  1/1      1              1            16m
root@Ubuntu-Server1$
root@Ubuntu-Server1$ kubectrl get pods
NAME            READY    STATUS    RESTARTS    AGE
myalpine       1/1     RUNNING   0            12m
root@Ubuntu-Server1$

```

If you need to delete a specific container, you can orchestrate it from the Master using the command given in Example 7-45. If the pod is using labels, it can also be deleted using the `kubectrl delete pods -l` command, as provided in Table 7-6.

Example 7-45 *Use Kubectl to Delete Nodes, Deployments, and Pods*

```

root@Ubuntu-Server1$
root@Ubuntu-Server1$ kubectl delete pods myalpine
pod "myalpine" deleted
root@Ubuntu-Server1$
root@Ubuntu-Server1$ kubectl get pods myalpine
Error from server (NotFound): pods "myalpine" not found
root@Ubuntu-Server1$
root@Ubuntu-Server1$ kubectl delete deployments alpine
deployment.extensions "alpine" deleted
root@Ubuntu-Server1$
root@Ubuntu-Server1$ kubectl get deployments
Error from server (NotFound): deployment.extensions "alpine" not found
root@Ubuntu-Server1$

```

To automate the instantiation, management, and deletion of pods and deployments, kubectl supports YAML, which plays a key role in deploying either a single instance of the objects or at scale. Chapter 8, “Application Developers’ Tools and Resources,” discusses the usage of JSON/XML and YAML.

Open Agent Container (OAC)

To support network device automation and management, Nexus switches can be enabled with Puppet and Chef agents. However, open agents cannot be directly installed on these platforms. To support these agents and similar applications, an isolated execution space within an LXC called the OAC was built.

As you see in Figure 7-11, the Open Agent Container (OAC) application is packaged into an .ova image and hosted at the same location where NX-OS images are published on Cisco.com.

Downloads Home / Switches / Data Center Switches / Nexus 7000 Series Switches / Nexus 7700 10-Slot Switch / Software on Chassis / NX-OS System Software- 8.3(1)

Nexus 7700 10-Slot Switch

Release 8.3(1)

My Notifications

Related Links and Documentation
[Release Notes for MDS NX-OS 8.3\(1\)](#)
[Release Notes for N7K NX-OS 8.3\(1\)](#)

File Information	Release Date	Size
Nexus 7700 Supervisor 3 System Software Image for 8.3(1) n7700-s3-dk9.8.3.1.bin	04-Jul-2018	516.28 MB
Open Virtualization Archive for the Cisco Systems Open Agent Container oac.8.3.1.ova	04-Jul-2018	44.54 MB

Figure 7-11 *Open Agent Container OVA Download*

First copy the .ova image to the Nexus switch. In Example 7-46, the file is copied to the bootflash file system in a Nexus 7700 switch.

OAC Deployment Model and Workflow

To install and activate OAC on your device, use the commands shown in Example 7-46. The virtual-service install command creates a virtual service instance, extracts the .ova file, validates the contents packaged into the file, validates the virtual machine definition, creates a virtual environment in the device, and instantiates a container.

Example 7-46 *Install OAC*

```
Nexus7700# virtual-service install name oac package bootflash:oac.8.3.1.ova
Note: Installing package 'bootflash:/oac.8.3.1.ova' for virtual service 'oac'. Once
the install has finished, the VM may be activated. Use 'show virtual-service list'
for progress
Nexus7700#
2019 Aug 28 10:22:59 Nexus7700 %VMAN-2-INSTALL_FAILURE: Virtual Service
[oac]::Install::Unpacking error::Unsupported OVA Compression/Packing format
2019 Aug 28 11:20:27 Nexus7700 %VMAN-5-PACKAGE_SIGNING_LEVEL_ON_INSTALL: Pack-
age 'oac.8.3.1.ova' for service container 'oac' is 'Cisco signed', signing level
allowed is 'Cisco signed'
2019 Aug 28 11:20:30 Nexus7700 %VMAN-2-INSTALL_STATE: Successfully installed virtual
service 'oac'
Nexus7700#
Nexus7700# show virtual-service list
Virtual Service List:
Name                Status              Package Name
-----
oac                  Installed           oac.8.3.1.ova
Nexus7700#
```

Using the **show virtual-service list** command, you can check the status of the container and make sure the installation is successful and the status is reported as **installed**. Then follow the steps given in Example 7-47 to activate the container. The NX-API feature is enabled, which will be used by OAC to perform the NX-OS CLIs directly from the container. As you see in the example, once the OAC is activated successfully, the **show virtual-service list** command shows the status of the container as **activating** and then **activated**.

Example 7-47 *Activate OAC*

```
Nexus7700# configure terminal
Nexus7700(config)# feature nxapi
Nexus7700(config)# virtual-service oac
Nexus7700(config-virt-serv)# activate
Nexus7700(config-virt-serv)# end
```

```
Note: Activating virtual-service 'oac', this might take a few minutes. Use 'show
virtual-service list' for progress.
Nexus7700#
Nexus7700# show virtual-service list
Virtual Service List:
Name                Status              Package Name
oac                 Activating         oac.8.3.1.ova
Nexus7700#
2019 Aug 28 11:23:06 Nexus7000 %$ VDC-1 %$ %VMAN-2-ACTIVATION_STATE: Successfully
activated virtual service 'oac'
Nexus7700#
Nexus7700# show virtual-service list
Virtual Service List:
Name                Status              Package Name
oac                 Activated          oac.8.3.1.ova
Nexus7700#
Nexus7700# 2019 Aug 28 11:23:06 Nexus7000 %$ VDC-1 %$ %VMAN-2-ACTIVATION_STATE:
Successfully activated virtual service 'oac'
```

As shown in Example 7-48, you can verify that the OAC is instantiated and actively running on the device with the **show virtual-service detail** command. The command supplies details of the resources allocated to the container, such as disk space, CPU, and memory.

Example 7-48 *Verify OAC Installation and Activation*

```
Nexus7000# show virtual-service detail
Virtual service oac detail
State                : Activated
Package information
Name                 : oac.8.3.1.ova
Path                 : bootflash:/oac.8.3.1.ova
Application
Name                 : OpenAgentContainer
Installed version    : 1.0
Description          : Cisco Systems Open Agent Container
Signing
Key type             : Cisco release key
Method               : SHA1
Licensing
Name                 : None
Version              : None
Resource reservation
Disk                 : 500 MB
Memory               : 384 MB
CPU                  : 1% system CPU
```

```
Attached devices
Type           Name           Alias
-----
Disk           _rootfs
Disk           /cisco/core
Serial/shell
Serial/aux
Serial/Syslog           serial2
Serial/Trace           serial3
```

Successful OAC activation depends on the availability of the required resources for OAC. If a failure occurs, the output of the `show virtual-service list` command will show the status as **Activate Failed** (see Example 7-49).

Example 7-49 *OAC Activation Failure*

```
Nexus7700# show virtual-service list
Virtual Service List:
Name           Status           Package Name
-----
oac            Activate Failed  oac.8.3.1.ova
Nexus7700#
```

To obtain additional information on the failure, you can use the `show system internal virtual-service event-history debug` command. As shown in Example 7-50, the reason for failure is clearly reported as insufficient disk space.

Example 7-50 *System Internal Event History*

```
Nexus7700# show system internal virtual-service event-history debug
243) Event:E_VMAN_MSG, length:124, at 47795 usecs after Wed Aug 28 09:23:52 2019
      (info): Response handle (nil), string Disk storage request (500 MB) exceeds
            remaining disk space (344 MB) on storage
244) Event:E_VMAN_MSG, length:74, at 47763 usecs after Wed Aug 28 09:23:52 2019
      (debug): Sending Response Message: Virtual-instance: oac - Response: FAIL
```

Instantiation of the OAC is persistent across the reload of the switch or supervisor engine. It means that the OAC will be instantiated upon supervisor engine reset or reload, but it will not be activated. It is not necessary to save the configurations with “copy running-config startup-config” to have the OAC instantiated and activated, without manual intervention, upon supervisor engine reset or reload. Because the OAC does not have high-availability support, the instantiation of the OAC is not replicated automatically to the standby supervisor engine. In other words, if you need to have OAC instantiated and activated for switchover, copy and save either the same .ova file or a different file in the standby supervisor engine’s bootflash.

Accessing OAC via the Console

To connect to the virtual service environment from the host Nexus switch, use the **virtual-service connect** command, as shown in Example 7-51.

Example 7-51 *Accessing OAC via the Console*

```
Nexus7700# virtual-service connect name oac console
Connecting to virtual-service. Exit using ^c^c^c
Trying 127.1.1.3...
Connected to 127.1.1.3.
Escape character is '^]'.

CentOS release 6.9 (Final)
Kernel 3.14.39ltsi+ on an x86_64

Nexus7700 login:
Password:
You are required to change your password immediately (root enforced)
Changing password for root.
(current) UNIX password:
New password:
Retype new password:
[root@Nexus7700 ~]#
[root@Nexus7700 ~]#whoami
root
[root@Nexus7700 ~]#
```

The default credentials to attach to the containers' console are **root/oac** or **oac/oac**. You must change the root password upon logging in. Just like in any other Linux environment, you can use Sudo to **root** after logging in as user **oac**.

Because you are accessing through console needs, you need to be on the switch first. The access can be slow, so many users prefer to access OAC via SSH. Before OAC can be accessed via SSH, the SSH service should be enabled and the container networking set up. The following section tells you how to enable this access method.

OAC Networking Setup and Verification

By default, networking in the OAC is done in the default routing table instance. Any additional route that is required (for example, a default route) must be configured natively in the host device and should not be configured in the container.

As you can see in Example 7-52, the **chvrf management** command is used to access a different routing instance (for example, the management VRF). After logging in to the container through the console, enable **SSH process/daemon (sshd)** in the **management VRF**.

Every VRF in the system has a numerical value assigned to it, so you need to make sure the `sshd` context matches the number assigned to the management VRF to confirm that the SSH process is active on the right VRF context. As shown in Example 7-52, the number assigned to VRF management is 2, which matches with the `DCOS_CONTEXT` assigned to the `SSHD` process.

Example 7-52 *Verify Container Networking*

```
[root@Nexus7700 ~]# chvrf management
[root@Nexus7700 ~]#
[root@Nexus7700 ~]# getvrf
management
[root@Nexus7700 ~]#
[root@Nexus7700 ~]# /etc/init.d/sshd start
Starting sshd:
[ OK ]
[root@Nexus7700 ~]#
[root@Nexus7700 ~]# more /etc/init.d/sshd | grep DCOS
export DCOS_CONTEXT=2
[root@Nexus7700 ~]#
[root@Nexus7700 ~]# vrf2num management
2
[root@Nexus7700 ~]# /etc/init.d/sshd status
openssh-daemon (pid 315) is running...
[root@Nexus7700 ~]#
```

Because NX-OS has allocated TCP port number 22 to the SSH process running in the host, configure an unused and different TCP port number for the OAC's SSH daemon. As demonstrated in Example 7-53, the `/etc/sshd_config` file has been edited to assign Port 2222 to OAC's SSH service, and the SSH service is listening for connections at 10.122.140.94, which is the `Mgmt0` interface of the Nexus switch.

Example 7-53 *Configure TCP Port for SSH*

```
[root@Nexus7700 ~]# cat /etc/ssh/sshd_config
<snip>
Port 2222
#AddressFamily any
ListenAddress 10.122.140.94
#ListenAddress ::
<snip>
[root@Nexus7700 ~]#

[root@Nexus7700 ~]#
```

Make sure to configure the DNS server and domain information so that OAC and agents installed in it can resolve domain names, as shown in Example 7-54.

Example 7-54 *Verify DNS Configuration*

```
[root@Nexus7700 ~]#
[root@Nexus7700 ~]# cat /etc/resolv.conf
nameserver 208.67.222.222
nameserver 208.67.220.220
[root@Nexus7700 ~]#
```

The command shown in Example 7-55 is performed in the host device, which confirms that a socket is open for OAC for the SSH connections at the IP address of the management port and TCP port 2222.

Example 7-55 *Verify Open Sockets*

```
Nexus7700# show sockets connection
Total number of netstack tcp sockets: 5
Active connections (including servers)
      Protocol State/      Recv-Q/   Local Address (port) /
              Context      Send-Q   Remote Address (port)
<snip>
[slxc]: tcp    LISTEN    0         10.122.140.94 (2222)
              default    0         * (*)
<snip>
Nexus7700#
```

Access the container to verify SSH accessibility, as shown in Example 7-56.

Example 7-56 *Verify SSH Access for OAC*

```
Ubuntu-Server1$
Ubuntu-Server1$ ssh -p 2222 root@10.122.140.94
CentOS release 6.9 (Final)
Kernel 3.14.39ltsi+ on an x86_64

Nexus7700 login: root
Password:
Last login: Tue Sep 10 10:26:46 on pts/0
#
```

If you are making changes to SSH parameters and settings in OAC, it is recommended that you restart the SSH service and check the status with `service sshd` commands. Now you have an active OAC that can be accessed via console or SSH. Next you will learn

how the kernel and OAC handles the packet from and to the front-panel ports in the host device, as illustrated in Figure 7-12.

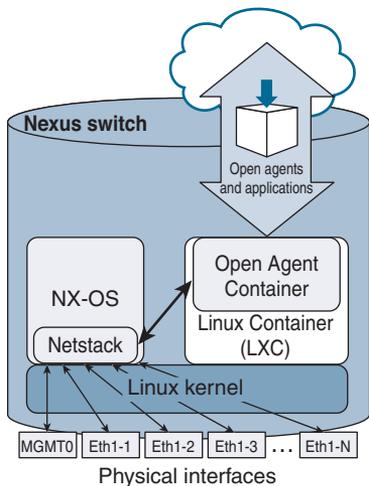


Figure 7-12 *Packet Handling in OAC*

As far as the containers are concerned, it all comes back to its namespace and the sockets and file descriptors associated to each container.

Once a socket is listening on a port, the kernel tracks those structures by namespace. As a result, the kernel knows how to direct traffic to the correct container socket. Here is a brief look at how traffic received by a Nexus switch’s front-panel port is forwarded to a specific container:

1. OAC implements a Message Transmission Service (MTS) tunnel to redirect container IP traffic to an NX-OS Netstack for forwarding lookup and packet processing to the front-panel port. This requires `libmts` and `libns` extensions, which are already included and set up in the `oac.ova`. Nexus 7000 has Netstack, which is a complete IP stack implementation in the user space of NX-OS. Netstack handles any traffic sent to the CPU for software processing.
2. The modified stack looks for the `DCOS_CONTEXT` environment variable, as mentioned in Example 7-56, to tag the correct VRF ID before sending the MTS message to Netstack.
3. The OAC is VDC aware because the implementation forwards traffic to the correct Netstack instance in which the OAC is installed.

Example 7-56 helped you verify that the OAC is accessible through SSH from an external device. In other words, the container should also be able to connect to the external network. Verify the reachability to the external network by sending ICMP pings to an external device, as shown in Example 7-57.

Example 7-57 *OAC Reachability to External Network*

```
[root@Nexus7700 ~]# chvrf management ping 10.122.140.65
PING 10.122.140.65 (10.122.140.65): 56 data bytes
64 bytes from 10.122.140.65: icmp_seq=0 ttl=254 time=2.495 ms
64 bytes from 10.122.140.65: icmp_seq=1 ttl=254 time=3.083 ms
64 bytes from 10.122.140.65: icmp_seq=2 ttl=254 time=2.394 ms^C
--- 10.122.140.65 ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 1.221/2.444/3.962/1.138 ms
[root@Nexus7700 ~]#
```

From within the OAC, in addition to accessing the network, the network administrator can access the device CLI using the **dohost** command, access the Cisco NX-API infrastructure, and more importantly, install and run Python scripts as well as 32-bit Linux applications.

Management and Orchestration of OAC

If there is a new version of OVA available for OAC, you can upgrade the currently active container using virtual-service commands, as shown in the following example. To upgrade, you need to deactivate the currently active container, as shown in Example 7-58.

Example 7-58 *Upgrade OAC*

```
Nexus7700(config)# virtual-service oac
Nexus7700(config-virt-serv)# no activate
Nexus7700(config-virt-serv)# end
2019 Sep  9 22:46:46 N77-A-Admin %$ VDC-1 %$ %VMAN-2-ACTIVATION_STATE: Successfully
deactivated virtual service 'oac'
Nexus7700#
Nexus7700# show virtual-service list
Virtual Service List:
Name                Status              Package Name
-----
oac                  Deactivated         oac.8.3.1.ova
Nexus7700#
Nexus7700# virtual-serv install name oac package bootflash:oac.8.3.1-v2.ova
Nexus7700#
Nexus7700# show virtual-service list
Virtual Service List:
Name                Status              Package Name
-----
oac                  Installed           oac.8.3.1-v2.ova
Nexus7700#
```

```
Nexus7700# config t
Nexus7700(config)# feature nxapi
Nexus7700(config)# virtual-service oac
Nexus7700(config-virt-serv)# activate
Nexus7700(config-virt-serv)# end
Nexus7700# show virtual-service list
Virtual Service List:
Name                Status              Package Name
-----
oac                  Activated           oac.8.3.1-v2.ova
Nexus7700#
```

To deactivate the container and uninstall the package, follow the steps as depicted in Example 7-59.

Example 7-59 *Deactivate OAC*

```
Nexus7700#
Nexus7700# config t
Nexus7700(config)# virtual-service oac
Nexus7700(config-virt-serv)# no activate
Nexus7700(config-virt-serv)# end
Nexus7700# show virtual-service list
Virtual Service List:
Name                Status              Package Name
-----
oac                  Deactivated         oac.8.3.1-v2.ova
Nexus7700#
Nexus7700# config t
Nexus7700(config)# no virtual service oac
Nexus7700(config)# exit
Nexus7700# virtual-service uninstall name oac
```

Installation and Verification of Applications

Open Agent Container, as the name suggests, is specifically developed to run open agents that cannot be natively run on NX-OS, such as Puppet agents and Chef agents.

Custom Python Application

To demonstrate the capability, you will look into a simple Python application. The Python file in Example 7-60 prints the system date and time every 10 seconds until the user stops the application by pressing Ctrl+C.

Example 7-60 *OAC—Sample Python Application*

```
[root@Nexus7700 ~]# more datetime.py

#!/usr/bin/python
import datetime
import time

while True:
    print("Time now is ... ")
    DateTime = datetime.datetime.now()
    print (str(DateTime))
    time.sleep(10)

[root@Nexus7700 ~]#
```

Check the file permissions and make sure the user **root** has permission to execute the file. Execute the Python file, as shown in Example 7-61.

Example 7-6 *Run Python Application in OAC*

```
[root@Nexus7700 ~]#
[root@Nexus7700 ~]# ls -l datetime.py
-rwxr--r-- 1 root root 194 Sep 10 23:16 datetime.py
[root@Nexus7700 ~]#
[root@Nexus7700 ~]# ./datetime.py
Time now is ...
2019-09-10 23:16:09.563576
Time now is ...
2019-09-10 23:16:19.573776
Time now is ...
2019-09-10 23:16:29.584028
^CTraceback (most recent call last):
  File "./ datetime.py", line 8, in <module>
    time.sleep(10)
KeyboardInterrupt
[root@Nexus7700 ~]#
```

Now that you know how to run a simple Python application in an OAC, you will see how to use Python APIs that are built-in and available in Nexus platforms. You can use these Python APIs to develop and run customized applications to monitor device health, track events, or generate alerts.

Application Using Python APIs

Cisco NX-OS has a built-in package providing API access to CLIs, both at the exec level as well as configuration commands, referred to as Python APIs. Example 7-62 is a simple Python script that leverages Python APIs that are natively available in the Nexus switches.

Example 7-62 Application Using Python APIs

```
[root@Nexus7700 ~]# more PY-API.py

#!/usr/bin/python
from cli import *
import json

print("STANDARD CLI OUTPUT ...")
print (cli('show interface brief'))

print("JSON FORMAT CLI OUTPUT ...")
print (clid('show interface brief'))

[root@Nexus7700 ~]#
```

Example 7-63 demonstrates the outputs generated by the application. As you notice, `cli` returns the raw format of the CLI results, including control and special characters. `clid` returns a dictionary of attribute names and values for the given CLI command.

Example 7-63 Run Python API Application in OAC

```
[root@Nexus7700 ~]# ls -l PY-API.py
-rwxr--r-- 1 root root 194 Sep 10 23:37 PY-API.py
[root@N77-A-Admin ~]#
[root@N77-A-Admin ~]# ./PY-API.py
STANDARD CLI OUTPUT ...
-----
Port    VRF      Status IP Address          Speed    MTU
-----
mgmt0   --       up      10.122.140.94       1000    1500
-----
JSON FORMAT CLI OUTPUT ...
{"TABLE_interface": {"ROW_interface": {"interface": "mgmt0", "state": "up", "ip_
  addr": "10.122.140.94", "speed": "1000", "mtu": "1500"}}}
[root@Nexus7700 ~]#
```

The `dohost` command in Example 7-64 is a Python wrapper script using NX-API functions and Linux domain sockets back to NX-OS. Using `dohost` capability, a user can perform `show` commands and configuration commands within the VDC in which the container is created.

Example 7-64 *Run NX-OS CLIs in OAC with dohost*

```
[root@N77-A-Admin ~]#
[root@N77-A-Admin ~]# dohost "show clock"
Time source is NTP
23:38:15.692 EST Tue Sep 10 2019
[root@N77-A-Admin ~]#
```

Package Management

As shown in Example 7-65, you can install packages in OAC using **yum install <package-name>** commands, just like in any CentOS Linux environment. Before installing packages, make sure to install them in the right VRF context. The namespace or VRF should have network connectivity and have the configurations required to resolve domain names.

Example 7-65 *OAC Package Management*

```
[root@Nexus7700 ~]# chvrf management yum install -y vim
Setting up Install Process
Resolving Dependencies
<snip>
```

Use **yum repolist** commands to verify the installed packages and repositories.

From OAC, you can run Open Agents, 32-bit Linux applications, and custom Python applications leveraging Python APIs, NX-APIs, or simple **dohost** commands to run CLIs and analyze the data. Chapter 9, “Container Deployment Use Cases,” will discuss the various use cases for packages and applications.

Summary

This chapter introduced the fundamentals of the NX-OS architecture—its key components and benefits. It presented the key built-in capabilities that enable application hosting and containers in Nexus switching platforms. It discussed various popular features such as Open Agent Containers (OAC), Guest Shell, Bash, and Docker containers—how to enable or instantiate them; how to configure them to communicate with external networks; how to enable console as well as SSH access; how to install simple Python or Python API-based applications and run those applications; how to instantiate Docker containers in supported platforms; and how to orchestrate them using Kubernetes.

Cisco NX-OS platforms have built-in capabilities helping developers deploy custom applications in the networking devices connected to the end hosts. These applications become more effective as they are brought closer to the data generated at the network edge, which can be processed in real time to gain insights. Above all, users can orchestrate and manage the lifecycle of containers and applications by activating or deactivating them,

upgrading the applications or packages installed, and leveraging the abilities the platforms provide for automation and scalability.

References

Cisco Nexus 7000 series Switches–NX-OS Architecture and Features: https://www.cisco.com/c/en/us/products/collateral/switches/nexus-7000-series-switches/white_paper_c11-622511.html

Open NX-OS and Linux–Developer Guide: <https://developer.cisco.com/docs/nx-os/>

Cisco Nexus 9000 series Python SDK User Guide and API Reference: <https://developer.cisco.com/docs/nx-os/#!cisco-nexus-9000-series-python-sdk-user-guide-and-api-reference>

Docker Documentation: <https://docs.docker.com/>

Alpine Linux–Home Page: <https://alpinelinux.org/>

Musl libc–Home Page: <http://musl.libc.org/>

Busybox–Home Page: <https://busybox.net/about.html>

Install and Setup Kubectl Guide: <https://kubernetes.io/docs/tasks/tools/install-kubectl/>

Kubectl Overview: <https://kubernetes.io/docs/reference/kubectl/overview/>

Index

Numbers

3GPP Cellular IoT (CIoT), 53
3rd Generation Partnership Project (3GPP), 64
5G wireless, 53
A10 Networks, 418
802.1Q, 24
802.1x, 239

A

AAA (authentication, authorization, and accounting), 320–321, 327
access control, 30
accessibility, verification of, 380, 383–384
ACLs (access control lists), 30–31, 239
active containers, listing, 90
active-active mode, 19
active-standby mode, 19
Address Resolution Protocol (ARP), 24
admin plane, 193
Alpine images
 instantiating Docker containers with, 263–266

container with Alpine image, 263–264
 docker attach command, 265
 file system mount, 265–266
 Linux container deployment, 223
anomaly detector application, 391–398
 floodlight application, 392–396
 docker create command, 393
 docker-compose.yml file, 393–394
 traffic capture, 394–395
 traffic classification, 395–396
 high-level procedure for, 392
 objectives of, 391
 running in NX-OS, 396–398
Ansible
 authentication, 348
 checking version of, 347
 hosts file, 347
 NETCONF operations with, 350–351
 overview of, 346
 playbooks, 348–350
 ansible.cfg file, 349
Apache NetBeans, 337

- Apache OpenWhisk, 17
- APIC, 42–44
- APIC-EM, 44
- APIs (application programming interfaces)
 - API-based applications in GuestShell, 254–256
 - definition of, 36
 - NX-API (Nexus API), 305–318
 - data management engine*, 309–310
 - enabling*, 306–309
 - managed objects*, 309–310
 - Message Format component*, 306
 - NX-API REST*, 310–318
 - NX-API Sandbox*, 313–315
 - overview of*, 305
 - Security component*, 306
 - Transport component*, 306
 - overview of, 36–37
 - Python, 297
 - in IOS-XE*, 302–305
 - in NX-OS*, 297–302
 - REST APIs
 - NX-API REST*, 310–318
 - overview of*, 38
 - SDN (software-defined networking), 33
- apk add --no-cache dhcp command, 366
- app hosting services, 405–406
 - Cisco Kinetic, 408
 - DNS (Domain Name System), 409
 - NetBeez Agent, 409–410
 - open-source applications, 410
 - OWAMP (One-Way Active Measurement Protocol), 407
 - perfSONAR, 408–409
 - Solenoid, 406
 - tcpdump, 407–408
 - TWAMP (Two-Way Active Measurement Protocol), 407
- AppGigEthernet interface, 125–126, 129–130
- application communication
 - application to external network, 98–99
 - application to host, 98
- application development
 - Cisco NX-SDK (Nexus Software Development Kit), 294–295
 - framework for, 62
- application hosting
 - CAF (Cisco Application Hosting Framework), 69–70
 - in IOS-XE, 146–149
 - Docker-style applications*, 152, 175
 - IOx application types*, 150–152
 - IOx framework*, 148–149
 - libvirt*, 146–148
 - LXC-based Guest Shell containers*, 150, 157–160
 - native Docker application in Catalyst 9300*, 182–186
 - PaaS-style applications*, 151, 161–166
 - virtual machine-based applications and hosting*, 151, 166–175
 - VMAN (Virtualization Manager)*, 146–148
 - in IOS-XR, 192–193
 - application hosting volumes*, 198–199
 - Docker-based application hosting*, 217–223
 - Linux-based application hosting*, 209–217
 - native application hosting*, 201–209
 - network configuration*, 216–217, 224–225
 - persistent application deployment*, 232–234
 - VRF namespace, application hosting in*, 226–232

application installation. *See also*
 deployment
 application deployment workflow, 156
 Chef, 355
 Cisco NX-SDK (Nexus Software Development Kit), 293–297
 Docker-style application on IOS-XE platforms, 182
 Guest Shell, 245–248, 253
 IOx, 157
 OAC (Open Agent Containers), 285–288
 application using Python APIs, 287–288
 custom Python application, 285–286
 package management, 288
 PaaS-style applications, 165–166
 Puppet, 351–353
 SDK, 207
 virtual machine-based applications, 172–175
Application Policy Infrastructure Controller, 42–44
application programming interfaces.
 See APIs (application programming interfaces)
application services, starting/stopping, 169
architecture, container, 15–17
ARP (Address Resolution Protocol), 24, 251, 252–253
ASICs (Doppler in Switching Platforms), 142
assurance, Cisco DNA Center, 46–47
Atom, 338
authentication
 AAA (authentication, authorization, and accounting), 320–321, 327
 in Ansible, 348
automated orchestration, 62
AVI Networks, 418
AVI Vantage, 418

AWS Cloud9, 338–339
AWS Lambda, 17
Azure Function, 17

B

bare metal as a service (BMaaS), 20
Base Header (NSH), 419
base-rootfs file, 390
Bash, 202, 256–260
 accessing from NX-OS, 257–258
 accessing via SSH, 258–260
 enabling, 256–257
 NX-OS support for, 240
Beacon, 35
Bell Labs, 2
BFD, Seamless (S-BFD), 384–391
 client, hosting on server, 390–391
 discriminators, 386
 Docker images, 388
 overview of, 385
 reflector sessions, 386–387
 reflectorbase, hosting on XR devices, 388–389
 as VNF (virtual network functions), 387–388
BGP (Border Gateway Protocol), 237, 395
Bidirectional Forwarding Detection. *See* S-BFD (Seamless BFD)
Big Switch Networks, Floodlight, 35
BIND configuration, 371
Bitbucket, 340–341
BMaaS (bare metal as a service), 20
Bootstrap, 340
Border Gateway Protocol (BGP), 237, 395
Bourne-Again Shell. *See* Bash
bridge networking
 configuration, 134–136
 definition of, 114

business drivers for virtualization, 3–6

- cost optimization, 5
- resilience, 5
- resource optimization, 4–5
- simplicity, 5

C**cache-coherent, non-uniform memory access (ccNUMA) machine, 3****CAF (Cisco Application Hosting Framework), 69–70****Cambridge Monitor System (CMS), 3****Cambridge Scientific Center, 3****capturing traffic, 394–395****Catalyst switches****Catalyst 9000**

DNS container installation in,
374–375

Docker container installation in,
368–369

DHCP Docker containers, 363–364

device license, 364

IOS-XE version verification,
363–364

IOX framework, enabling, 364

native Docker application in Catalyst 9300, 182–186

Docker image creation, 182–186

Docker pull and export, 182–183

non-uniform memory access) machine, 3**CCP (Cisco Container Platform), 416–417****CDP (Cisco Discovery Protocol), 332, 395****certificates, NX-API**

- exporting, 308–309
- generating, 307–308

cgroups, 16**Chef**

- cookbooks, creating, 354–355
- installation, 355
- overview of, 346
- resources, 354

chef-client command, 356–357**chkconfig utility, 267****chvrf management command, 280****CIoT (3GPP Cellular IoT), 53****Cisco APIC, 42–44****Cisco APIC-EM, 44****Cisco Application Hosting Framework (CAF), 69–70****Cisco ASA 5506 firewall, 10****Cisco Container Platform (CCP), 416–417****Cisco Digital Network Architecture (DNA), 55****Cisco Discovery Protocol (CDP), 332, 395****Cisco DNA Center, 45–47, 55**

assurance, 46–47

design, 45

policy, 46

provision, 46

Cisco Elastic Services Controller (ESC), 49**Cisco Enterprise NFV Open Ecosystem, 418****Cisco Guest Shell. *See* Guest Shell****Cisco Inter-Switch Link (ISL), 24****Cisco IOS-XE. *See* IOS-XE platforms****Cisco IOS-XR. *See* IOS-XR platforms****Cisco IOS-XR platforms. *See* IOS-XR platforms****Cisco Kinetic, 52–53, 408****Cisco native-app hosting network model, 106–111**

dedicated networking

Cisco IOS-XE configuration,
125–130

- Cisco IOS-XR configuration*, 131
 - Nexus OS configuration*, 131
- shared networking
 - Cisco IOS-XE configuration*, 115–117
 - Cisco IOS-XR configuration*, 117–122
 - Cisco Nexus OS configuration*, 122–125
 - output*, 107–108
 - overview of*, 106–107
 - support matrix for*, 125
- Cisco NFV infrastructure architecture, 411–412
- Cisco NX-OS. *See* NX-OS platforms
- Cisco NX-SDK (Nexus Software Development Kit), 291–297
- Cisco Open Agent Containers. *See* OAC (Open Agent Containers)
- Cisco Open SDN Controller, 36
- Cisco SDN solutions
 - Cisco APIC, 42–44
 - Cisco APIC-EM, 44
 - Cisco DNA Center, 45–47
 - assurance*, 46–47
 - design*, 45
 - policy*, 46
 - provision*, 46
 - modern network design with, 47
- Cisco Service Containers, 67–69
 - CLI-based commands, 68
 - OVA file for, 67–68
 - software support matrix for, 69
- Cisco TrustSec (CTS), 239
- Cisco Ultra Services Platform, 53–54, 415–416
- Cisco Unified Computing Servers (UCS), 10, 412
- Cisco Unified Fabric, 239
- Cisco User Plane Function (UPF), 416
- Cisco Virtual Internet Routing Lab (VIRL), 7
- ciscobridge, 103
- cisco-centos virtual machine
 - application installation in, 210–211
 - environment preparation, 214
 - launching, 210
 - libvirtd daemon, 214
 - LXC container root filesystem, 211–215
 - LXC spec file, 213–214
 - virsh command, 215
 - virsh console command, 215
- Cisco-IOS-XR-cdp-cfg.yang, 332
- clear configuration lock command, 330
- clear NETCONF-yang session command, 329–330
- cli command, 255, 287, 303–304
- cli import command, 299–300
- cli Python module
 - cli command, 303–304
 - clip command, 303–304
 - configure command, 304
 - configurep command, 304
 - displaying details about, 303–304
 - execute command, 305
 - executep command, 305
- clid command, 255, 287
- clients
 - Chef, 355
 - Cisco Application Hosting Framework (CAF), 69
 - Docker, 86–87, 261
 - S-BFD (Seamless BFD), 390–391
- clip command, 303–304
- cloud computing, 13
- Cloud Native Computing Foundation (CNCF), 79
- Cloud9, 338–339

- cloud-native network functions (CNFs), 63–65, 416
- cloud-native reference model, 61–62
- CMS (Cambridge Monitor System), 3
- CMTs (Configuration Management Tools)
 - Ansible
 - authentication*, 348
 - checking version of*, 347
 - hosts file*, 347
 - NETCONF operations with*, 350–351
 - overview of*, 346
 - playbooks*, 348–350
 - Chef
 - cookbooks, creating*, 354–355
 - installation*, 355
 - overview of*, 346
 - resources*, 354
 - Puppet
 - installation*, 351–353
 - OSPF configuration with*, 353–354
 - overview of*, 346, 351
- CNCF (Cloud Native Computing Foundation), 79
- CNFs (cloud-native network functions), 63–65, 416
- CNI (Container Network Interface) model
 - components of, 114–115
 - deployment, 136
- CNM (Container Network Model), 111–114
 - components of, 111–112
 - network creation, 113
 - network types, 112–114
 - defining*, 113
 - Docker Bridge*, 114, 134–136
 - Docker Host*, 113, 132–134
 - Docker MACVLAN*, 114
 - Docker None*, 113, 131–132
 - Docker Overlay*, 114
- output, 112
- communication modes, 97–99
- compute platforms
 - Cisco UCS (Unified Computing Servers), 412
 - ENCS (Enterprise Network Compute System), 412
- computer evolution, history of, 1–2
- confd, 327, 328
- config.ini file, 399, 404
- Configuration Management Tools. *See* CMTs (Configuration Management Tools)
- configure command, 304
- configure terminal command, 364
- configurep command, 304
- Connected Vehicles, 53–54
- Container Network Interface (CNI) model
 - components of, 114–115
 - deployment, 136
- Container Network Model. *See* CNM (Container Network Model)
- Container Runtime, 63
- Container Runtime Interface (CRI), 63
- Context Header (NSH), 419
- control groups, 16
- Control Plane and User Plane Separation (CUPS), 64, 415
- control plane health check
 - anomaly detector application
 - floodlight application*, 392–396
 - high-level procedure for*, 392
 - objectives of*, 391
 - running in NX-OS*, 396–398
 - overview of, 362, 391–398
- Control Plane Policing (CoPP) counters, 399–401
- control plane virtualization, 33–34, 64
- Control Program (CP) hypervisor, 3
- controllers (SDN), 34–36
 - open source, 35–36
 - OpenFlow, 34–35

cookbooks (Chef), 354–355
 cookies, `nxapi_auth`, 306
 CoPP (Control Plane Policing) counters, 399–401
 copy running-config startup-config command, 395
 <copy-config> operation, 329
 cost optimization, 5
 counters, CoPP (Control Plane Policing), 399–401
 CP (Control Program) hypervisor, 3
 CPU share, 198–199
 CRI (Container Runtime Interface), 63
 CTERA, 418
 CTS (Cisco TrustSec), 239
 CUPS (Control Plane and User Plane Separation), 64, 415
 curl command, 273, 322–323, 326, 380, 383

D

daemons

Docker

running, 91–92, 263
verification of, 85–86

`libvirtd`, 214

Data Management Engine (DME), 306

data management engine (NX-API), 309–310

Data Model Interface (DMI), 327

data modeling languages

table of, 39

YANG, 39–42

db.foo.bar.local file, 371

dedicated networking

Cisco IOS-XE configuration, 125–130

AppGigEthernet interface, 125–126

Layer 2 mode, 129–130

numbered routing, 126–128

unnumbered routing, 128–129

VirtualPortGroup interface, 125–126

Cisco IOS-XR configuration, 131

Nexus OS configuration, 131

output, 109–111

overview of, 108–109

default namespaces, 122, 124, 251

default VRFs, 248

define command, 233

DELETE operations (RESTCONF), 320

demilitarized zone (DMZ), 55

deployment

Cisco NX-SDK (Nexus Software Development Kit), 293

CNFs (cloud-native network functions), 63

Docker

container, running, 89–91

daemon, running, 91–92

daemon status verification, 85–86

Docker Client, 86–87

images, pulling from registry to local store, 87–89

Guest Shell, 159–160, 243–245

guestshell enable command, 243

guestshell resize commands, 245

show guestshell command, 243–245

status and resource allocation, 243–245

HAProxy Docker container, 382

IOS-XE deployment workflow, 156

Kubernetes

deploying workload with, 94–95

master nodes, enabling, 92–93

workers nodes, enabling, 93–94

LXC (Linux containers), 81–84

CLI commands, 82–83

- Guestshell rootsfs content*, 83–84
- IOx Service on IOS-XE*, 81–82
- IOX-specific syslog message*, 83
- in Ubuntu server*, 84
- Node.js Docker container, 378
- OAC (Open Agent Containers), 277–279
 - activation*, 277–278
 - failures in*, 279
 - installation*, 277
 - system internal event history*, 279
 - verification*, 278–279
- use cases, 361
 - configuration consistency check*, 362
 - control plane health check*, 362
 - DHCP Docker container deployment*, 363–369
 - DNS Docker container deployment*, 369–375
 - HAProxy and Node containers*, 375–384
 - hardware/software stability check*, 362
 - infrastructure for proof-of-concept and testing purposes*, 363
 - inventory management*, 361
 - operational data monitoring*, 363
 - resource usage and scalability check*, 362
 - traffic profiling and top talkers*, 363
- virtual network services, 48
- design, network.** *See network design*
- development environment**
 - IOx PaaS-style applications, 161
 - PaaS-style applications, 161
- development tools**
 - Ansible
 - authentication*, 348
 - checking version of*, 347
 - hosts file*, 347
 - NETCONF operations with*, 350–351
 - overview of*, 346
 - playbooks*, 348–350
 - Apache NetBeans, 337
 - Atom, 338
 - AWS Cloud9, 338–339
 - Bitbucket, 340–341
 - Bootstrap, 340
 - Chef
 - cookbooks, creating*, 354–355
 - installation*, 355
 - overview of*, 346
 - resources*, 354
 - Docker Build, 343
 - Docker Hub, 344
 - account creation*, 344
 - docker pull command*, 345
 - DTR (Docker Trusted Registry)*, 345
 - repository*, 344
 - Dockerfile, 342–343
 - Eclipse, 339–340
 - GitHub, 337–338
 - Linx, 336–337
 - NETCONF Agent
 - Ansible with*, 350–351
 - in IOS-XE*, 327–331
 - in IOS-XR*, 331–332
 - in NX-OS*, 333–336
 - overview of*, 318–319
 - RESTCONF compared to*, 320
 - Node.js, 341
 - NX-API (Nexus API), 305–318
 - data management engine*, 309–310
 - enabling*, 306–309
 - managed objects*, 309–310
 - Message Format component*, 306
 - NX-API REST*, 310–318

- NX-API Sandbox*, 313–315
 - overview of*, 305
 - Security component*, 306
 - Transport component*, 306
- NX-SDK (Nexus Software Development Kit), 291–297
 - deployment modes*, 293
 - framework*, 291–292
 - installation and activation*, 293–297
 - release versions*, 292
- Puppet
 - installation*, 351–353
 - OSPF configuration with*, 353–354
 - overview of*, 346, 351
- Python APIs (application programming interfaces)
 - in IOS-XE*, 302–305
 - in NX-OS*, 297–302
- RESTCONF Agent
 - in IOS-XE*, 320–323
 - NETCONF compared to*, 320
 - in NX-OS*, 323–326
 - overview of*, 318–319
- Zend Studio, 339
- device licensing, verification of, 364
- device virtualization, history of, 26–29
- DevNet Data Model, 310
- DFA (Dynamic Fabric Automation), 239
- DHCP (Dynamic Host Configuration Protocol), 24, 185
 - DHCP snooping, 239
 - DHCPd, 363, 365
- DHCP Docker container deployment, 363–369
 - Catalyst switch configuration, 363–364
 - device license*, 364
 - IOS-XE version verification*, 363–364
 - IOX framework, enabling*, 364
 - container creation, 364–368
 - DHCPd configuration*, 365
 - Dockerfile file*, 366–367
 - folders*, 365
 - image, building*, 367–368
 - image archive*, 368
 - container installation in Catalyst 9000, 368–369
 - dhcpd.conf file*, 365–366
- Digital Network Architecture (DNA), 55
- DISCO, 3
- discriminators, S-BFD (Seamless BFD), 386
- distinguished names (DNs), 309
- DME (Data Management Engine), 306
- DMI (Data Model Interface), 327
- dmiauthd, 328
- DMZ (demilitarized zone), 55
- DNA (Digital Network Architecture), 55
- DNA Center (DNAC), 45–47, 55
 - assurance, 46–47
 - design, 45
 - policy, 46
 - provision, 46
- DNA-Advantage licensing, 185
- DNs (distinguished names), 309
- DNS (Domain Name System), 409
 - NFV (network functions virtualization) and, 47
 - verifying configuration of, 282
- DNS Docker container deployment, 369–375
 - container creation, 373–374
 - container installation in Catalyst 9000, 374–375
 - preparation for, 370–373
 - BIND configuration*, 371
 - Dockerfile file*, 371–373
 - folders*, 370

- nameserver daemon configuration*, 370
- Docker, 75–78. *See also* Dockerfile
 - architecture, 75–76
 - CNM (Container Network Model), 111–114
 - components of*, 111–112
 - Docker Bridge*, 114, 134–136
 - Docker Host*, 113, 132–134
 - Docker MACVLAN*, 114
 - Docker None*, 113, 131–132
 - Docker Overlay*, 114
 - network creation*, 113
 - output*, 112
 - DHCP Docker container deployment, 363–369
 - Catalyst switch configuration*, 363–364
 - container creation*, 364–368
 - container installation in Catalyst 9000*, 368–369
 - DNS Docker container deployment, 369–375
 - container creation*, 373–374
 - container installation in Catalyst 9000*, 374–375
 - preparation for*, 370–373
 - Docker Build, 343
 - Docker Hub, 344–345
 - Docker-based application hosting, 217–223
 - container deployment workflow*, 223
 - loading from public registry*, 218–220
 - loading images from local registry*, 220–222
 - loading images from public registry*, 218–220
 - loading manually to local store*, 222–223
 - HAProxy container solution, 375–384
 - Docker image creation*, 377–378, 381–382
 - HAProxy Docker container installation*, 382–384
 - HAProxy load balancer setup*, 380–381
 - project initiation*, 375–376
 - Web server configuration*, 376–377
 - Web server Docker container installation*, 378–380
 - hosting architecture, 196–198
 - images
 - building*, 341–344, 367–368, 373, 377, 381
 - Docker-style application on IOS-XE platform*, 179–181
 - HAProxy and Node containers*, 377–378, 381–382
 - image archive*, 368, 373–374, 378, 382
 - loading from local registry*, 220–222
 - loading manually to local store*, 222–223
 - native Docker application in Catalyst 9300*, 182–186
 - in NX-OS platforms*, 262
 - PaaS-style applications on IOS-XE*, 162
 - publishing*, 344–345
 - pulling*, 87–89, 182–183, 218–220
 - S-BFD (Seamless BFD)*, 387–388
 - installation, 294
 - native application in Catalyst 9300, 182–186
 - deployment*, 184–185
 - Docker container networking*, 185
 - Docker image creation*, 182–183
 - Docker pull and export*, 182–183
 - licensing requirements*, 185–186

- network stack, 90–91
- in NX-OS platforms, 260–276
 - architecture*, 260–261
 - Docker client*, 261
 - Docker daemon*, 263
 - Docker hosts*, 261–262
 - instantiating with Alpine image*, 263–266
 - managing*, 266–268
 - NX-OS Docker health check*, 398–404
- orchestration with Kubernetes, 268–276
 - architecture*, 268–270
 - Kubectl*, 273–276
 - Masters, building*, 270–273
- overview of, 80
- software support matrix for, 77–78
- toolchain, 175
- docker attach alpine command**, 90
- docker attach command**, 265
- Docker Bridge networking**
 - configuration, 134–136
 - definition of, 114
- Docker Build**, 343
- docker build command**, 373
- docker create command**, 393
- Docker Host networking**
 - configuration, 132–134
 - definition of, 113
- Docker Hub**, 344
 - account creation, 344
 - repository, 344
 - docker pull command*, 345
 - DTR (Docker Trusted Registry)*, 345
- docker images command**, 223
- docker load command**, 88, 223
- docker logs <container-name> command**, 367
- Docker MACVLAN network type**, 114
- docker network ls command**, 113
- Docker None networking**
 - configuration, 131–132
 - definition of, 113
- Docker Overlay networking**, 114
- docker ps command**, 90, 221, 223
- docker pull command**, 88, 184, 221, 222, 293, 345
- docker run command**, 89, 221, 223, 268
- docker save alpine command**, 88, 222
- docker stop command**, 267–268
- Docker Trusted Registry (DTR)**, 345
- docker version command**, 197
- docker-compose up command**, 396–398, 401–404
- docker-compose.yml file**, 393–394, 396–398, 399, 401–404
- Dockerfile**, 76–77, 342–343
 - creating folders for, 376, 380
 - editing, 366–367, 371–373, 376–377, 381, 388
- dockerpart file**, 263
- Docker-style applications**
 - Docker-style application on IOS-XE platforms
 - caveats and restrictions*, 175
 - components of*, 152
 - development workflow*, 177
 - Docker toolchain*, 175
 - images, creating*, 179–180
 - installing and running*, 182
 - IOx package creation with YAML*, 180–181
 - package repository*, 177–178
 - Python application development*, 178
 - on IOS-XE platforms, 175
- dohost command**, 255, 284, 287, 288
- Domain Name System**. *See* DNS (Domain Name System)
- Doppler in Switching Platforms (ASICs)**, 142

Dot1Q, 24–25
 DTR (Docker Trusted Registry), 345
 Dynamic Fabric Automation (DFA), 239
 Dynamic Host Configuration Protocol (DHCP), 24, 185

E

EC2 (Elastic Compute Cloud), 3
 Eclipse, 339–340
 Edge and Fog Processing Module (EFM), 408
 edit-config operator, 335
 <edit-config> operation, 329
 EEM (Embedded Event Manager), 297
 EIGRP (Enhanced Interior Gateway Routing Protocol), 395
 Elastic Compute Cloud (EC2), 3
 Elastic Services Controller (ESC), 49
 Electronic Numerical Integrator and Computer (ENIAC), 2
 Embedded Event Manager (EEM), 297
 Embedded Services Processor (ESP), 144
 ENCS (Enterprise Network Compute System), 412
 endpoints, Docker, 112
 Enhanced Interior Gateway Routing Protocol (EIGRP), 395
 Enhanced SerDes Interconnect (ESI), 145
 ENIAC (Electronic Numerical Integrator and Computer), 2
 Enterprise File Service Platform, 418
 Enterprise Network Compute System (ENCS), 412
 Enterprise NFV Open Ecosystem, 418
 enterprise virtualization, 30–31
 EOBC (Ethernet Out-of-Band Channel), 145
 EPC (Evolved Packet Core) network, 64
 ESC (Elastic Services Controller), 49
 ESI (Enhanced SerDes Interconnect), 145

ESP (Embedded Services Processor), 144
 /etc/ansible/hosts file, 349
 etcd, 80
 /etc/ssh/sshd_config file, 246
 /etc/sysconfig/docker file, 263
 /etc/yum/yum.conf file, 203
 Ethernet Out-of-Band Channel (EOBC), 145
 ETL (Extract, Transform, and Load), 17
 ETSI (European Telecommunication Standards Institute), 47
 Evolved Packet Core (EPC) network, 64
 exaBGP, 406
 execute command, 305
 executep command, 305
 expected traffic, 391
 exporting certificates, 308–309
 Extensible Markup Language (XML), 38, 306
 external networks, namespaces to, 100–102
 namespace connectivity, 101
 namespace creation, 100
 veth interface, 100–101
 Extract, Transform, and Load (ETL), 17

F

FaaS (Function as a Service), 17–18, 421–423
 FabricPath, 239
 failure detection, S-BFD (Seamless BFD) for, 384–391
 client, 390–391
 discriminators, 386
 Docker images, 388
 overview of, 385
 reflector sessions, 386–387
 reflectorbase, 388–389
 as VNF (virtual network functions), 387–388

fault detection, 237–238
 FCoE, 239
 FD.IO, 420
 feature bash-shell command, 256–257
 feature nxapi command, 306–307
 FED (Forwarding Engine Driver), 142
 file systems, virtual machine, 169–170
 firewalls

- Cisco ASA 5506, 10
- ONEFW (One Firewall), 67

 Fission, 423
 Floodlight, 35
 floodlight application, 392–396

- docker create command, 393
- docker-compose.yml file, 393–394
- running in NX-OS, 396–398
- traffic capture, 394–395
- traffic classification, 395–396

 FMAN (Forwarding Manager), 142
 folders

- creating, 376, 380
- Docker, 365
- HAProxy, 375–376
- Node.js, 375–376

 FortiGate firewalls, 418
 Fortinet, 418
 Forwarding Engine Driver (FED), 142
 Forwarding Manager (FMAN), 142
 full virtualization, 6–7
 Function as a Service (FaaS), 17–18, 421–423
 fwd_ew interface, 121
 fwdintf interface, 121

G

GDT (global description table), 6
 generic routing encapsulation (GRE), 30–31

GET requests

- NX-API REST, 317–318
- RESTCONF Agent, 320
 - in IOS-XE*, 322–323
 - in NX-OS*, 325–326

 GigabitEthernet1 interface, 116
 git command, 293
 GitHub, 337–338
 global description table (GDT), 6
 global VRF route table, 121
 global-vrf namespace, 119–120, 121
 GRE (generic routing encapsulation), 30–31
 guest enable command, 83
 guest operating system (guest OS), 14
 Guest Shell, 70–71

- deploying with LXC, 81–84
 - CLI commands*, 82–83
 - Guestshell rootsfs content*, 83–84
 - IOx Service on IOS-XE*, 81–82
 - IOX-specific syslog message*, 83
 - in Ubuntu server*, 84
- in IOS-XE, 157–160
 - accessing*, 160
 - Guest Shell container activation*, 159–160
 - IOx activation*, 157
 - network configuration*, 157–158
 - verification of*, 159–160
- native-app hosting with dedicated networking, 127–128
- in NX-OS, 239–240, 242–256
 - accessing*, 245–248
 - application installation*, 245–248, 253
 - deployment model and workflow*, 243–245
 - network configuration*, 248–253
 - OVA file*, 242

Python application development,
253–256

support and resource
requirements, 239–240

with shared networks, 124–125

versions of, 71

guestshell command, 246

guestshell destroy command, 83

guestshell disable command, 83

guestshell enable command, 82–84, 159,
243

guestshell reboot command, 245

guestshell resize commands, 245

guestshell run command, 82–84, 160

guestshell sync command, 83

H

HAProxy, 375–384

Docker container installation, 382–384

Docker image creation, 377–378, 381–
382

load balancer setup, 380–381

project initiation, 375–376

Web server configuration, 376–377

Web server Docker container installation,
378–380

hardware-assisted virtualization, 7

hardware/software stability check, 362

HEAD operations, 320

health check, Docker, 398–404

config.ini file, 399

Control Plane Policing (CoPP) counters,
399–401

docker-compose.yml file, 399

high-level procedure for, 392–398

objectives of, 398

running, 401–404

Helium, 35–36

help command, 297, 303

high availability, 18

Hollerith, Herman, 1–2

Hollerith Cards, 1–2

hosted hypervisors, 10

hosting

Docker-style application on IOS-XE, 175

caveats and restrictions, 175

development workflow, 177

Docker toolchain, 175

images, creating, 179–180

installing and running, 182

IOx package creation with YAML,
180–181

package repository, 177–178

Python application development,
178

environment readiness for

IOS-XR platforms, 198

NX-OS platforms, 239–241

PaaS-style applications, 161–166

components of, 151

development environment, 161

Docker image creation, 162

installing and running, 165–166

IOx package creation with YAML,
162–165

Python application development,
161–162

supported platforms, 161

S-BFD clients, 390–391

S-BFD reflectorbase, 388–389

virtual machine-based applications,

166–175

components of, 151

installing and running, 172–175

IOx package creation with YAML,
170–172

network configuration, 174

tool/library installation, 168–169

virtual machine file systems,

169–170

virtual machines, building, 167–168

hosts file, Ansible, 347
 Hot Standby Router Protocol (HSRP), 395
 HSRP (Hot Standby Router Protocol), 395
 HTTPS (Hypertext Transfer Protocol Secure)
 NX-API support for, 306
 Transport Layer Security (TLS)-based, 321
 Hydrogen, 35–36
 Hyperkube, 92–93
 Hypertext Transfer Protocol Secure. *See* HTTPS (Hypertext Transfer Protocol Secure)
 hypervisors
 hosted, 10
 native, 10

I

I2C (inter-integrated circuit), 145
 IaaS (infrastructure as a service), 20
 IBM Cambridge Scientific Center, 3
 IBN (Intent-Based Networking), 55, 57
 IES (Industrial Ethernet Switches), 55
 ietf-interfaces modules, 40–41
 images (Docker)
 building, 341–344, 367–368, 373, 377, 381
 Docker-style application on IOS-XE platforms, 179–180
 HAProxy, 377–378, 381–382
 image archive, 368, 373–374, 378, 382
 native Docker application in Catalyst 9300, 182–186
 PaaS-style applications on IOS-XE, 162
 publishing, 344–345
 pulling, 87–89, 182–183
 S-BFD (Seamless BFD), 387–388
 index.js file, editing, 376-C09.5564
 Industrial Ethernet Switches (IES), 55
 Information Technology Infrastructure Library (ITIL) framework, 4
 infrastructure as a service (IaaS), 20
 infrastructure virtualization, history of, 23–27
 init process, 100
 in-service software upgrade (ISSU), 40
 installation. *See also* deployment
 application deployment workflow, 156
 Chef, 355
 Cisco NX-SDK (Nexus Software Development Kit), 293–297
 Docker-style application on IOS-XE platforms, 182
 Guest Shell, 245–248, 253
 IOx, 157
 OAC (Open Agent Containers), 285–288
 application using Python APIs, 287–288
 custom Python application, 285–286
 package management, 288
 PaaS-style applications, 165–166
 Puppet, 351–353
 SDK, 207
 virtual machine-based applications, 172–175
 instantiating containers, 265–266
 with Docker, 263–266
 container with Alpine image, 263–264
 daemon, 85–86, 91–92
 docker attach command, 265
 Docker Client, 86–87
 pulling images for, 87–89
 running containers, 89–91
 with LXC (Linux containers), 81–84
 S-BFD (Seamless BFD) reflector container, 389
 Intent-Based Networking (IBN), 55
 interface sync, 193

interfaces

AppGigEthernet, 125–126, 129–130

VirtualPortGroup, 125–126

inter-integrated circuit (I2C), 145

Internet of Things. *See* IoT (Internet of Things)

Internet Systems Consortium, 363

interprocess communication (IPC), 81, 143

Inter-Switch Link (ISL), 24

inventory management use case, 361

invoke function, 17

IO Privilege Level (IOPL) flags, 6

IOPL (IO Privilege Level) flag, 6

ios_vrf module, 348

IOS-XE platforms. *See also* use cases

Ansible

authentication, 348

checking version of, 347

hosts file, 347

NETCONF operations with, 350–351

overview of, 346

playbooks, 348–350

application deployment workflow, 156

application hosting capabilities, 146–149

IOx application types, 150–152

IOx framework, 148–149

libvirt, 146–148

VMAN (Virtualization Manager), 146–148

Cisco IOS-XE CLI, 185

dedicated networking in, 125–130

AppGigEthernet interface, 125–126

Layer 2 mode, 129–130

numbered routing, 126–128

unnumbered routing, 128–129

VirtualPortGroup interface, 125–126

Docker-style application, 175

caveats and restrictions, 175

components of, 152

development workflow, 177

Docker toolchain, 175

images, creating, 179–180

installing and running, 182

IOx package creation with YAML, 180–181

package repository, 177–178

Python application development, 178

history of, 140–142

key components of, 139–140

LXC-based Guest Shell containers, 157–160

components of, 150

Guest Shell container activation, 159–160

IOx activation, 157

network configuration, 157–158

native Docker application in Catalyst

9300, 182–186

deployment, 184–185

Docker container networking, 185

Docker image creation, 182–186

Docker pull and export, 182–183

licensing requirements, 185–186

NETCONF Agent, 327–331

enabling, 327–329

NETCONF-Yang processes in, 328–329

operations with, 329–331

supported features, 331

operation states, 156

- PaaS-style applications, 161–166
 - components of*, 151
 - development environment*, 161
 - Docker image creation*, 162
 - installing and running*, 165–166
 - IOx package creation with YAML*, 162–165
 - Python application development*, 161–162
 - supported platforms*, 161–166
- Python APIs, 302–305
 - accessing*, 302
 - cli command*, 303–304
 - clip command*, 303–304
 - configure command*, 304
 - configurep command*, 304
 - displaying details about*, 303
 - execute command*, 305
 - executep command*, 305
- releases, 140–141
- resource requirements, 153–156
 - memory*, 153–154
 - storage*, 153–154
 - VirtualPortGroup*, 154–155
 - vNICs (virtual network interface cards)*, 155–156
- RESTCONF Agent, 320–323
 - enabling*, 320–321
 - operations with*, 322–323
 - supported methods*, 320–321
- routing platforms, 144–146
- shared networking, 115–117
- switching platforms, 142–144
- virtual machine-based applications, 166–175
 - components of*, 151
 - installing and running*, 172–175
 - IOx package creation with YAML*, 170–172
 - network configuration*, 174
 - tool/library installation*, 168–169
 - virtual machine file systems*, 169–170
 - virtual machines, building*, 167–168
- IOS-XR platforms. See also use cases**
 - architecture
 - application hosting architecture*, 192–193
 - application hosting volumes*, 198–199
 - CPU share*, 198–199
 - Docker hosting architecture*, 196–198
 - hosting environment readiness*, 198
 - KIM (Kernel Interface Module)*, 193–195
 - memory*, 200–201
 - network namespaces*, 195–196
 - resource allocation*, 201
 - software evolution and*, 190–191
 - storage*, 198–199
 - Chef
 - cookbooks, creating*, 354–355
 - installation*, 355
 - overview of*, 346
 - resources*, 354
 - container management, 232–234
 - dedicated networking, 131
 - Docker-based application hosting, 217–223
 - container deployment workflow*, 223
 - loading from public registry*, 218–220
 - loading images from local registry*, 220–222

- loading images from public registry, 218–220*
- loading manually to local store, 222–223*
- evolution of, 190–191
- hosting S-BFD reflectorbase on, 388–389
- Linux-based application hosting, 209–217
 - application installation in test virtual machine, 210–211*
 - environment preparation, 214*
 - libvirtd daemon, 214*
 - LXC container root filesystem, 211–215*
 - LXC spec file, 213–214*
 - TPNNS namespace login, 212–213*
 - virsh command, 215*
 - virsh console command, 215*
 - virtual machine, launching, 210*
 - workflow summary, 209–210*
- native application hosting, 201–209
 - building RPM file for, 206–209*
 - from existing RPM file, 202–206*
 - overview of, 201–202*
- NETCONF Agent, 331–332
 - enabling, 331*
 - operations with, 332*
- network configuration, 216–217, 224–225
 - name resolution, 224–225*
 - network reachability, 224–225*
 - proxy configuration, 225*
- persistent application deployment, 232–234
- shared networking, 117–122
 - global VRF route table, 121*
 - LXC spec file, 122*
 - TPA configuration, 121*
 - TPNNS (third-party network namespace), 119, 121–122*
 - XRNNS (XR network namespace), 117–119*
- VRF namespace, application hosting in, 226–232
 - with LXC (Linux containers), 229–232*
 - VRF namespace configuration, 226–229*
- IOS-XRv 9000, instantiating, 414–415**
- IoT (Internet of Things), 51–57**
 - Cisco Kinetic, 52–53, 408
 - Cisco Ultra Services Platform, 53–54
 - Connected Vehicles, 53–54
 - devices in, 51–57
 - Edge Computing, 55
 - manufacturing use case, 55–57
- IOx framework**
 - application hosting capabilities, 148–149
 - application types, 150–152
 - architecture, 148–149
 - enabling, 364
 - IOx Service on IOS-XE, 81–82
 - IOX-specific syslog messages, 83
 - LXC-based Guest Shell containers, 157
 - Guest Shell container activation, 159–160*
 - network configuration, 157–158*
 - package creation with YAML, 162–165, 170–172, 180–181
 - virtual machine-based applications and hosting, 170–172
- IP Address Assignment and Management (IPAM), 98**
- ip link add vethcisco01 type veth command, 100–101**
- ip netns add cisco1 command, 100**
- ip netns exec global-vrf ip link command, 196**
- IP Source Guard, 239**
- ip unnumbered command, 128**
- IP_Move application, 293–297**
 - activating, 295–296
 - development of, 294–295
 - Linux environment, building, 293–294

- packaging, 295
- running, 295–296
- IPAM (IP Address Assignment and Management), 98
- IPC (interprocess communication), 81, 143
- iPerf application
 - installing from YUM repository, 205–206
 - installing with Chef, 355
 - RPM spec file, 208–209
- iperf3 command, 205–206
- IPv6 (SRv6) data plane, 420–421
- ISCSI (Small Computer System Interface over IP), 239
- ISL (Inter-Switch Link), 24
- ISSU (in-service software upgrade), 40
- ITIL (Information Technology Infrastructure Library) framework, 4

J

- JavaScript
 - creating folders for, 376
 - index.js file, editing, 376-C09.5564
- JSON (JavaScript Object Notation), 38, 255
 - NX-API support for, 306
 - Python API automation with JSON results, 299–300

K

- K8S. *See* Kubernetes
- Kernel Interface Module (KIM), 193–195
- Kernel Sockets, 247–248
- Kernel-based Virtual Machine (KVM), 146
- keys, Puppet, 351
- <kill-session> request, 330
- KIM (Kernel Interface Module), 193–195

- Kinetic, 52–53, 408
- kstack networking, 247
- Kube Controller Manager, 80
- Kube-apiserver, 80
 - /kube/config file, 274
 - kubeconfig file, 274
- Kubectl, 273–276
- kubectl cluster-info command, 93
- kubectl command, 94
- kubectl get pod -o wide command, 95
- Kubeless, 423
- Kubelet, 80
- Kube-proxy, 81
- Kubernetes, 79–81, 268–276
 - architecture, 79, 268–270
 - CNI (Container Network Interface) model
 - components of*, 114–115
 - deployment*, 136
 - deploying workload with, 94–95
 - Docker daemon, running, 91–92
 - Kubectl, 273–276
 - masters, 80, 92–93, 270–273
 - serverless computing and, 423
 - worker nodes, 80–81, 93–94
- Kube-scheduler, 80
- KVM (Kernel-based Virtual Machine), 146
- kvm command, 414–415

L

- LACP (Link Aggregation Control Protocol), 27
- Lambda, 17
- libnetwork package, 111
- library installation, IOS-XE, 168–169
- libvirt, 146–148, 192
- libvirtd daemon, 214

- licensing requirements, native Docker application in Catalyst 9300, 185–186
- Link Aggregation Control Protocol (LACP), 27
- Link Layer Discovery Protocol (LLDP), 395
- Linux bridge, creation of, 103
- Linux containers. *See* LXC (Linux containers)
- Linux Foundation, OpenDaylight, 35–36
- Linux KVM, 10
- Linux-based application hosting, 209–217
 - application installation in test virtual machine, 210–211
 - environment preparation, 214
 - libvirtd daemon, 214
 - LXC container root filesystem, 211–215
 - LXC spec file, 213–214
 - TPNNS namespace login, 212–213
 - virsh command, 215
 - virsh console command, 215
 - virtual machine, launching, 210
 - workflow summary, 209–210
- Linux, 336–337
- LIPC (Local IPC), 143
- LLDP (Link Layer Discovery Protocol), 395
- lo interface, 121
- load balancers
 - HAProxy, 380–381
 - NFV (network functions virtualization) and, 47
- Local IPC (LIPC), 143
- local registries, loading Docker images from, 220–222
- local stores, loading Docker images to, 218–223
 - from local registry, 220–222
 - manually, 222–223
 - from public registry, 218–220
- <lock> operation, NETCONF Agent, 329–330
- Lubuntu, 7
- LXC (Linux containers), 16–17, 66–67, 242
 - CAF (Cisco Application Hosting Framework), 69–70
 - daemons
 - running*, 91–92
 - status verification*, 85–86
 - Docker Client, 86–87
 - Guest Shell, 70–71
 - deploying with LXC*, 81–84
 - overview of*, 70
 - versions of*, 71
 - images, pulling, 87–89
 - instantiation and management, 81–84
 - LXC-based Guest Shell containers, 213–214
 - components of*, 150
 - Guest Shell container activation*, 159–160
 - IOx activation*, 157–160
 - network configuration*, 157–158
 - OAC (Open Agent Containers), 71–75
 - commands*, 74–75
 - OAC.OVA file*, 71–72
 - oac.xml file*, 72–74
 - resource output*, 74–75
 - root filesystems in, 211–215
 - running, 89–91
 - active containers, listing*, 90
 - Docker network stack*, 90–91
 - docker run command*, 89
 - Service Containers, 67–69
 - CLI-based commands*, 68
 - OVA file for*, 67–68
 - software support matrix for*, 69
 - spec files, 122, 213–214

TPNNS namespace login, 212–213
 VRF namespace, application hosting in,
 226–229

M

Macvlan, 114
 Maestro, 36
 managed objects, 309–310
 management information tree (MIT),
 309–310
 management namespace, 122–124,
 252–253
 management VRFs, 248
 manifest files
 Cisco OAC (Open Agent Containers),
 72–74
 Cisco Service Containers, 67
 MANO (management and orchestration).
See orchestration
 manufacturing use case, 55–57
 masters (Kubernetes), 80, 92–93,
 270–273
 MEC (mobile edge computing), 64
 memory
 in IOS-XE, 153–154
 in IOS-XR, 200–201
 Message Format component (NX-API),
 306
 Message Queue IPC (MQIPC), 144
 Message Transmission Services (MTS),
 237
 /misc/app_host volume, 198
 MIT (management information tree),
 309–310
 mobile edge computing (MEC), 64
 models, network. *See* network models
 Moore, Gordon, 2
 MPLS (SR-MPLS), 420–421
 MPLS VPN (Multiprotocol Label
 Switching VPN), 25–27

MQIPC (Message Queue IPC), 144
 MTS (Message Transmission Services),
 237
 Multiprotocol Label Switching VPN
 (MPLS VPN), 25–27
 multitenancy, 19–20

N

name resolution, 224–225
 named.conf file, 370
 nameserver daemon configuration, 370
 namespaces
 to another namespace, 100–102
 Linux bridge, 103
 namespace creation, 102
 veth interface, 102–104
 Cisco NXOS, 105
 definition of, 16, 99
 to external network, 100–102
 namespace connectivity, 101
 namespace creation, 100
 veth interface, 100–101
 Guest Shell, 248–250
 IOS-XR platforms, 195–196
 key points for, 104–105
 NAS (network-attached storage), 12, 239
 NAT (Network Address Translation), 47
 National Institute of Standards and
 Technology (NIST), 13
 native application hosting, IOS-XR
 platforms, 201–209
 building RPM file for, 206–209
 from existing RPM file, 202–206
 overview of, 201–202
 native Docker application in Catalyst
 9300, 182–186
 deployment, 184–185
 Docker container networking, 185
 Docker image creation, 182–186

- Docker pull and export, 182–183
- licensing requirements, 185–186
- native hypervisors, 10**
- native-app hosting network model, 106–111**
 - dedicated networking
 - Cisco IOS-XE configuration, 125–130*
 - Cisco IOS-XR configuration, 131*
 - Nexus OS configuration, 131*
 - output, 109–111*
 - overview of, 108–109*
 - shared networking
 - Cisco IOS-XE configuration, 115–117*
 - Cisco IOS-XR configuration, 117–122*
 - Cisco Nexus OS configuration, 122–125*
 - output, 107–108*
 - overview of, 106–107*
 - support matrix for, 125*
- ncsshd, 328**
- ndbmand, 328**
- Neon, 35–36**
- nesd, 328**
- nested virtualization, 7**
- NetBeez Agent, 409–410**
- NETCONF Agent, 39**
 - Ansible with, 350–351
 - in IOS-XE, 327–331
 - enabling, 327–329*
 - NETCONF-Yang processes in, 328–329*
 - operations with, 329–331*
 - supported features, 331*
 - in IOS-XR, 331–332
 - enabling, 331*
 - operations with, 332*
 - in NX-OS, 333–336
 - enabling, 333*
 - operations with, 333–336*
 - overview of, 318–319
 - RESTCONF compared to, 320
- NETCONF-yang command, 327, 328**
- netns_identify command, 203**
- Netscout, 418**
- Netstack module, 237**
- Network Address Translation. *See* NAT (Network Address Translation)**
- NETwork CONFIguration (NETCONF), 39**
- network connectivity, 99**
- network design**
 - Cisco DNA Center, 45
 - with NFV (network functions virtualization), 47
 - with SDN (software-defined networking), 47
 - virtualization
 - high availability, 18*
 - multitenancy, 19–20*
 - workload distribution, 19*
- network device virtualization, history of, 26–29**
- Network Function Virtualization Infrastructure Software (NFVIS), 412**
- network functions virtualization. *See* NFV (network functions virtualization)**
- network infrastructure virtualization, history of, 23–27**
- Network Interface Module (NIM), 153**
- network isolation, 99**
- Network Management Servers (NMS), 404**
- network measurement**
 - OWAMP (One-Way Active Measurement Protocol), 407
 - perfSONAR, 408–409
 - TWAMP (Two-Way Active Measurement Protocol), 407

network models, 105

- Cisco native-app hosting
 - dedicated networking, 108–111*
 - shared networking, 106–111*
 - Docker CNM (Container Network Model), 111–114
 - components of, 111–112*
 - Docker Bridge, 114, 134–136*
 - Docker Host, 113, 132–134*
 - Docker MACVLAN, 114*
 - Docker None, 113, 131–132*
 - Docker Overlay, 114*
 - network creation, 113*
 - network types, 112–114*
 - output, 112*
 - Kubernetes CNI (Container Network Interface)
 - components of, 114–115*
 - deployment, 136*
- network namespaces**
- to another namespace, 100–102
 - Linux bridge, 103*
 - namespace creation, 102*
 - veth interface, 102–104*
 - Cisco NXOS network namespace, 105
 - definition of, 99
 - to external network, 100–102
 - namespace connectivity, 101*
 - namespace creation, 100*
 - veth interface, 100–101*
 - IOS-XR platforms, 195–196
 - key points for, 104–105
- network operational service applications, 405**
- app hosting services, 405–406
 - Cisco Kinetic, 408*
 - DNS (Domain Name System), 409*
 - NetBeez Agent, 409–410*
 - open-source applications, 410*

- OWAMP (One-Way Active Measurement Protocol), 407*
 - perfSONAR, 408–409*
 - Solenoid, 406*
 - tcpdump, 407–408*
 - TWAMP (Two-Way Active Measurement Protocol), 407*
- Cisco NFV offerings
- CCP (Cisco Container Platform), 416–417*
 - Cisco UCS (Unified Computing Servers), 412*
 - Cisco Ultra Service Platform, 415–416*
 - consolidated view of, 417–418*
 - ENCS (Enterprise Network Compute System), 412*
 - NFV infrastructure architecture, 411–412*
 - virtual routers and switches, 414–415*
- containers and service chaining, 418–421
- NSH (Network Service Header), 419–420*
 - SRv6 (Segment Routing v6), 420–421*
- serverless computing, 421–423
- network programmability. See programmability**
- Network Service Header (NSH), 419–420**
- Network Services Orchestrator (NSO), 49**
- network-attached storage (NAS), 12, 239**
- networking, 105. See also SDN (software-defined networking); virtualization**
- communication modes, 97–99
 - consistency checks, 362
 - dedicated, 131
 - Cisco IOS-XE, 125–130*
 - Cisco IOS-XE configuration, 125–130*

- Cisco IOS-XR*, 131
- Nexus OS*, 131
- Nexus OS configuration*, 131
- output*, 109–111
- overview of*, 108–109
- Docker CNM (Container Network Model), 111–114
 - components of*, 111–112
 - Docker Bridge*, 114, 134–136
 - Docker Host*, 113, 132–134
 - Docker MACVLAN*, 114
 - Docker None*, 113, 131–132
 - Docker Overlay*, 114
 - network creation*, 113
 - output*, 112
- Kubernetes CNI (Container Network Interface) model, 136
 - components of*, 114–115
 - deployment*, 136
- namespaces
 - to another namespace*, 100–102
 - Cisco NXOS network namespace*, 105
 - definition of*, 99
 - to external network*, 100–102
 - key points for*, 104–105
- network slicing, 64
- network stack, 90–91
- SANs (storage area networks), 12
- shared, 106–107, 108
 - Cisco IOS-XE*, 115–117
 - Cisco IOS-XR*, 117–122
 - Cisco Nexus OS*, 122–125
 - support matrix for*, 125
- Nexus 9000 switches, enabling as Kubernetes worker node, 93–94
- Nexus API. *See* NX-API (Nexus API)
- Nexus Software Development Kit. *See* NX-SDK (Nexus Software Development Kit)
- Nexus Software Development Kit (NX-SDK), 291–297
- NFV (network functions virtualization)
 - app hosting services, 405–406
 - Cisco Kinetic*, 408
 - DNS (Domain Name System)*, 409
 - NetBeez Agent*, 409–410
 - open-source applications*, 410
 - OWAMP (One-Way Active Measurement Protocol)*, 407
 - perfSONAR*, 408–409
 - Solenoid*, 406
 - tcpdump*, 407–408
 - TWAMP (Two-Way Active Measurement Protocol)*, 407
 - Cisco NFV offerings
 - CCP (Cisco Container Platform)*, 416–417
 - Cisco NFV infrastructure architecture*, 411–412
 - Cisco UCS (Unified Computing Servers)*, 412
 - Cisco Ultra Service Platform*, 415–416
 - consolidated view of*, 417–418
 - ENCS (Enterprise Network Compute System)*, 412
 - virtual routers and switches*, 414–415
 - Cisco SDN solutions
 - Cisco APIC*, 42–44
 - Cisco APIC-EM*, 44
 - Cisco DNA Center*, 45–47
 - definition of, 11
 - elements in, 48
 - modern network design with, 47
 - virtual network service deployment, 48
- NFVI (NFV Infrastructure), 48, 411
- NFVIS (Network Function Virtualization Infrastructure Software), 412
- NFVO (NFV Orchestration), 411

- nginx process**
 - NETCONF Agent and, 328
 - NX-API (Nexus API), 306
 - RESTCONF Agent and, 321
- Nicira Networks, NOX, 35**
- NIM (Network Interface Module), 153**
- NIST (National Institute of Standards and Technology), 13**
- NMS (Network Management Servers), 404**
- Node.js, 341**
 - folders, 375–376
 - HAProxy and Node containers, 375–384
 - Docker image creation, 377–378, 381–382*
 - HAProxy Docker container installation, 382–384*
 - HAProxy load balancer setup, 380–381*
 - project initiation, 375–376*
 - Web server configuration, 376–377*
 - Web server Docker container installation, 378–380*
- nodes, managing with Kubectl, 275–276**
- none networking (Docker), 113, 131–132**
- noninteractive mode, Python APIs in, 300–302**
- NOX, 35**
- NSH (Network Service Header), 419–420**
- NSO (Network Services Orchestrator), 49**
- numbered routing, 126–128**
 - Guest Shell, 127–128
 - VirtualPortGroup configuration, 127
- NX-API (Nexus API), 305–318**
 - data management engine, 309–310
 - enabling, 306–309
 - certificates, exporting, 308–309*
 - certificates, generating, 307–308*
 - feature nxapi command, 306–307*
 - managed objects, 309–310
 - Message Format component, 306
 - NX-API REST, 310–318
 - logical view, 311*
 - NX-API GET request in Python v3, 317–318*
 - NX-API Sandbox, 313–315*
 - response to REST API request, 312–313*
 - URL format, 311*
 - NX-API Sandbox, 313–315
 - overview of, 305
 - Security component, 306
 - Transport component, 306
- nxapi sandbox command, 313**
- nxapi_auth, 306**
- NX-OS platforms. *See also* use cases**
 - anomaly detector application, 391–398
 - floodlight application, 392–396*
 - high-level procedure for, 392*
 - objectives of, 391*
 - running, 396–398*
 - Bash, 256–260
 - accessing from NX-OS, 257–258*
 - accessing via SSH, 258–260*
 - enabling, 256–257*
 - benefits of, 238–239
 - Cisco NXOS network namespace, 105
 - Docker containers, 260–276
 - architecture, 260–261*
 - Docker client, 261*
 - Docker daemon, 263*
 - Docker hosts, 261–262*
 - instantiating with Alpine image, 263–266*
 - managing, 266–268*

- orchestration with Kubernetes,*
268–276
- foundation of, 235–238
- Guest Shell, 242–256
 - accessing,* 245–248
 - application installation,* 253
 - deployment model and workflow,*
243–245
 - network configuration,*
248–253
 - OVA file,* 242
 - Python application development,*
253–256
- hosting environment readiness, 239–241
 - Bash,* 240
 - Guest Shell,* 239–240
 - OAC (Open Agent Containers),*
240–241
- NETCONF Agent, 333–336
 - enabling,* 333
 - operations with,* 333–336
- NX-OS Docker health check, 398–404
 - config.ini file,* 399
 - Control Plane Policing (CoPP)*
counters, 399–401
 - docker-compose.yml file,* 399
 - high-level procedure for,*
392–398
 - objectives of,* 398
 - running,* 401–404
- OAC (Open Agent Containers), 276–288
 - accessing via console,* 280
 - application installation and verification,*
285–288
 - architecture,* 276–277
 - deactivating,* 285
 - deployment model and workflow,*
277–279
 - network configuration,*
280–284
 - upgrading,* 284–285
- Puppet
 - installation,* 351–353
 - OSPF configuration with,* 353–354
 - overview of,* 346, 351
- Python APIs in, 297–302
 - benefits of,* 302
 - noninteractive mode,* 300–302
 - Python API package, displaying,*
297–299
- RESTCONF Agent in, 323–326
 - enabling,* 323–325
 - operations with,* 325–326
- shared networking, 122–125
 - default network namespace,* 122,
124
 - Guest Shell,* 124–125
 - management network namespace,*
122–124
- software architecture, 235–238
- NX-SDK (Nexus Software Development Kit), 291–297
 - deployment modes, 293
 - framework, 291–292
 - installation and activation, 293–297
 - activation,* 295–296
 - application development,* 294–295
 - Linux environment, building,*
293–294
 - packaging applications,* 295
 - running applications,* 295–296
 - release versions, 292

O

- OAC (Open Agent Containers), 71–75,
276–288
 - accessing via console, 280
 - application installation and verification,
285–288
 - application using Python APIs,*
287–288

- custom Python application*, 285–286
 - package management*, 288
- architecture, 276–277
- commands, 74–75
- deactivating, 285
- deployment model and workflow, 277–279
 - failures in*, 279
 - OAC activation*, 277–278
 - OAC installation*, 277
 - OAC verification*, 278–279
 - system internal event history*, 279
- network configuration, 280–284
 - DNS (Domain Name System)*, 282
 - OAC reachability to external network*, 284
 - open sockets*, 282
 - SSH access*, 282
 - TPC port*, 281
 - verification of*, 281
- NX-OS support for, 240–241
- OAC.OVA file, 71–72
- oac.xml file, 72–74
- resource output, 74–75
- upgrading, 284–285
- OBFL (On-Board Failure Logging), 139
- objects, managed, 309–310
- OCI (Open Container Initiative), 63
- On-Board Failure Logging (OBFL), 139
- ONEFW (One Firewall), 67
- One-Way Active Measurement Protocol (OWAMP), 407
- ONF (Open Network Foundation), 35
- ONOS (Open Source Network OS), 36
- Open Agent Containers. *See* OAC (Open Agent Containers)
- Open Container Initiative (OCI), 63
- Open Network Foundation (ONF), 35
- Open SDN Controller, 36
- Open Shortest Path First (OSPF), 27, 237, 353–354, 395–396
- open sockets, verification of, 282
- open source controllers, 35–36
- Open Source Network OS (ONOS), 36
- Open Virtual Switch (OVS), 420
- Open Virtualization Format (OVF), 14
- OpenDaylight, 35–36
- OpenFlow, 34–35, 420
- open-source tools
 - Apache NetBeans, 337
 - app hosting services, 410
 - Atom, 338
 - AWS Cloud9, 338–339
 - Bitbucket, 340–341
 - Bootstrap, 340
 - Eclipse, 339–340
 - GitHub, 337–338
 - Linux, 336–337
 - Node.js, 341
 - Zend Studio, 339
- openssl commands, 307–308
- OpenTransit, 36
- OpenWhisk, 17
- operating expenses (OpEx), 33
- operation states (IOS-XE), 156
- operational data monitoring, 363
- OpEx (operating expenses), 33
- opkg utility, 178
- optimization
 - cost, 5
 - resource, 4–5
- orchestration
 - application development framework, 62
 - automated, 62
 - cloud-native reference model, 61–62
 - CNFs (cloud-native network functions), 63–65, 416

- deployment model*, 63
- framework*, 64
- Container Runtime, 63
- Docker, 75–78
 - architecture*, 75–76
 - CNM (Container Network Model), 111–114
 - container, running*, 89–91
 - daemon, running*, 91–92
 - daemon status verification*, 85–86
 - Docker Client*, 86–87
 - Dockerfile*, 76–77
 - images, pulling from registry to local store*, 87–89
 - network stack*, 90–91
 - software support matrix for*, 77–78
- in IOS-XE platforms
 - application deployment workflow*, 156
 - application hosting capabilities*, 146–149
 - Docker-style application*, 140–142, 175–182
 - key components of*, 139–140
 - LXC-based Guest Shell containers*, 157–160
 - native Docker application in Catalyst 9300*, 182–186
 - operation states*, 156
 - PaaS-style applications*, 161–166
 - resource requirements*, 153–156
 - routing platforms*, 144–146
 - switching platforms*, 142–144
 - virtual machine-based applications and hosting*, 166–175
- in IOS-XR platforms
 - architecture*, 192–201
 - container management*, 232–234
 - Docker-based application hosting*, 217–223
 - evolution of*, 190–191
 - Linux-based application hosting*, 209–217
 - native application hosting*, 201–209
 - network configuration*, 216–217, 224–225
 - persistent application deployment*, 232–234
 - VRF namespace, application hosting in*, 226–232
- Kubernetes, 79–81
 - architecture*, 79
 - CNI (Container Network Interface) model, 114–115
 - deploying workload with*, 94–95
 - deployment*, 136
 - Docker daemon, running*, 91–92
 - master nodes*, 80
 - master nodes, enabling*, 92–93
 - worker nodes*, 80–81
 - workers nodes, enabling*, 93–94
- LXC (Linux containers)
 - CAF (Cisco Application Hosting Framework)*, 69–70
 - Cisco Guest Shell*, 70–71, 81–84
 - Cisco OAC (Open Agent Containers)*, 71–75
 - Cisco Service Containers*, 67–69
 - container instantiation and management with*, 81–84
 - overview of*, 66–67
- NFV (network functions virtualization) and, 47
- in NX-OS platforms
 - Bash*, 256–260
 - benefit of NX-OS*, 238–239
 - Docker containers*, 260–276
 - Guest Shell*, 242–256

- hosting environment readiness, 239–241*
- software architecture, 235–238*
- overview of, 65–66
- OSPF (Open Shortest Path First), 27, 237, 353–354, 395–396
- OTV (Overlay Transport Virtualization), 237
- OVA file, 242
- overlay networking, 114
- Overlay Transport Virtualization (OTV), 237
- OVF (Open Virtualization Format), 14
- OVS (Open Virtual Switch), 420
- OWAMP (One-Way Active Measurement Protocol), 407

P

- PaaS (platform as a service)
 - components of, 151
 - PaaS-style applications, 161–166
 - components of, 151*
 - development environment, 161*
 - Docker image creation, 162*
 - installing and running, 165–166*
 - IOx package creation with YAML, 162–165*
 - Python application development, 161–162*
 - supported platforms, 161*
- packages
 - Cisco NX-SDK (Nexus Software Development Kit), 295
 - Docker repository, 177–178
 - IOx
 - Docker-style application on IOS-XE, 180–181*
 - PaaS-style applications on IOS-XE, 162–165*
 - virtual machine-based applications and hosting, 170–172*
 - OAC (Open Agent Containers), 288
 - Python API, 297–299
 - package.yaml descriptor file
 - Docker-style applications, 176
 - PaaS-style application, 163
 - virtual machine file system, 170–171
 - packet capture, 407–408
 - Packet Gateway (PGW), 64
 - PAM (Programmable Authentication Module), 306
 - paravirtualization, 7
 - PATCH operations, 320
 - path isolation, 30
 - .pcapng file, 394–395
 - perfSONAR, 408–409
 - persistence
 - Docker, 266–267
 - persistent application deployment, 232–234
 - PSS (Persistent Storage System), 237
 - PGW (Packet Gateway), 64
 - ping command, 101
 - playbooks, Ansible
 - example of, 348–349
 - running, 349
 - POAP (Power On Auto Provisioning), 253, 297
 - pods, 275–276
 - policy, Cisco DNA Center, 46
 - ports, TCP, 281
 - POST requests, 320
 - Power On Auto Provisioning (POAP), 253, 297
 - POX, 35
 - programmability, 38–42
 - data modeling languages
 - table of, 39*
 - YANG, 39–42
 - REST APIs, 38

Programmable Authentication Module (PAM), 306
 proof-of-concept, infrastructure for, 363
 provision, Cisco DNA Center, 46
 proxy configuration, 225
 PSS (Persistent Storage System), 237
 public registries, pulling Docker images from, 218–220
 publishing Docker images, 344–345
Puppet
 installation, 351–353
 OSPF configuration with, 353–354
 overview of, 346, 351
PUT operations, 320
Python APIs, 297. See also Python application development
 application using, 287–288
 in IOS-XE, 302–305
 accessing, 302
 cli command, 303–304
 clip command, 303–304
 configure command, 304
 configurep command, 304
 displaying details about, 303
 execute command, 305
 executep command, 305
 in NX-OS, 297–302
 benefits of, 302
 noninteractive mode, 300–302
 Python API package, displaying, 297–299
Python application development, 38, 317–318. See also Python APIs
 Docker-style application on IOS-XE, 178
 in Guest Shell in NX-OS, 253–256
 API-based applications, 254–256
 running applications, 253–254
 IOx PaaS-style applications, 161–162
 OAC (Open Agent Containers)

application using Python APIs, 287–288
 installation, 285–286
 package management, 288
python command, 253–254, 300–302

Q-R

QEMU (Quick Emulator), 146, 169
RADIUS
 NETCONF Agent, 327
 RESTCONF Agent, 320–321
 reachability, network, 224–225
RedHat Package Manager (RPM), 202
 reflect function, 388
 reflector sessions (S-BFD), 386–387
 reflectorbase (S-BFD), 388–389
 register_disc function, 388
 registries
 DTR (Docker Trusted Registry), 345
 loading Docker images from
 from local, 220–222
 from public, 218–220
 relative names (RNs), 310
Remote IPC (RIPC), 143
repositories
 Docker Hub, 344
 YUM
 configuration, 203
 default, 203
 installing native applications from, 204
representational state transfer (REST) APIs, 38
requests
 GET, 317–318, 322–323, 325–326
 NX-API REST, 312–316
resilience, 5
resolv.conf file, 224–225

- resource optimization, 4–5
 - resource requirements, 19
 - IOS-XE, 153–156
 - memory*, 153–154
 - storage*, 153–154
 - VirtualPortGroup*, 154–155
 - vNICs (virtual network interface cards)*, 155–156
 - IOS-XR, 201
 - resource usage and scalability check, 362
 - REST (representational state transfer)
 - APIs
 - NX-API REST, 310–318
 - logical view*, 311
 - NX-API GET request in Python v3*, 317–318
 - NX-API Sandbox*, 313–315
 - response to REST API request*, 312–313
 - URL format*, 311
 - overview of, 38. *See also* RESTCONF Agent
 - RESTCONF Agent
 - in IOS-XE, 320–323
 - enabling*, 320–321
 - operations with*, 322–323
 - supported methods*, 320–321
 - NETCONF compared to, 320
 - in NX-OS, 323–326
 - enabling*, 323–325
 - operations with*, 325–326
 - overview of, 318–319
 - RIB (Routing Information Base) table, 406
 - RIPC (Remote IPC), 143
 - RNs (relative names), 310
 - role-based access control (RBAC), 302
 - rootfs, 83–84, 179
 - route command, 224
 - route injection agents, Solenoid, 406
 - Route Processor (RP), 144
 - routing
 - global VRF route table, 121
 - Guest Shell
 - default namespace*, 251
 - management namespace*, 252–253
 - IOS-XE routing platforms, 144–146
 - RIB (Routing Information Base) table, 406
 - route configuration
 - numbered*, 126–128
 - unnumbered*, 128–129
 - route sync, 193
 - virtual routers and switches, 414–415
 - RP (Route Processor), 144
 - RPM (RedHat Package Manager) files, 202
 - IP_Move application, 295
 - native hosting from
 - existing RPM files*, 202–206
 - newly built RPM files*, 206–209
 - RPM spec file, 208–209
 - rpm_gen.py, 295
 - rpmbuild -ba command, 209
 - run bash command, 117–118, 257–258
 - run guestshell python command, 253–254
 - runc, 63
 - Ryu, 36
-
- ## S
- SaaS (software as a service), 20
 - SAE (Secure Agile Exchange), 50
 - sandbox, 111–112, 313–315
 - SANs (storage area networks), 12
 - S-BFD (Seamless BFD), 384–391
 - client, hosting on server, 390–391
 - discriminators, 386

- Docker images, 388
- overview of, 385
- reflector sessions, 386–387
- reflectorbase, hosting on XR devices, 388–389
- as VNF (virtual network functions), 387–388
- Scapy**, 395
- SDA (Software-Defined Access)**, 55
- SDK installation**, 207
- SDN (software-defined networking)**
 - APIs (application programming interfaces), 33
 - Cisco SDN solutions
 - Cisco APIC*, 42–44
 - Cisco APIC-EM*, 44
 - Cisco DNA Center*, 44
 - control plane, 33
 - controllers, 34–36
 - open source*, 35–36
 - OpenFlow*, 34–35
 - enablers, 33
 - APIs (application programming interfaces)*, 36–37
 - control plane virtualization*, 33–34
 - programmability*, 38–42
 - SDN controllers*, 34–36
 - high-level architecture of, 32–33
 - IoT (Internet of Things), 51–57
 - Cisco Kinetic*, 52–53
 - Cisco Ultra Services Platform*, 53–54
 - devices in*, 51–57
 - manufacturing use case*, 55–57
 - modern network design with, 47
- Seamless BFD**. *See* S-BFD (Seamless BFD)
- Secure Agile Exchange (SAE)**, 50
- Secure Shell**. *See* SSH (Secure Shell)
- Security component (NX-API)**, 306
- Segment Routing (SRv6)**, 420–421
- self-signed certificates**
 - exporting, 308–309
 - generating, 307–308
- sequential command persistence**, 299
- server virtualization**, 8–10
- serverless computing**, 17–18, 421–423
- service chaining**, 418–421
 - NSH (Network Service Header), 419–420
 - SRv6 (Segment Routing v6), 420–421
- Service Containers**, 67–69
 - CLI-based commands, 68
 - OVA file for, 67–68
 - software support matrix for, 69
- service docker command**, 86
- service docker start command**, 91
- service docker status command**, 92
- service function chaining (SFC)**, 64
- Service Gateway (SGW)**, 64
- Service mesh**, 62
- Service Oriented Architecture (SOA)**, 13
- Service Path Header (NSH)**, 419
- service providers**, 31
- Service Set Identifiers (SSIDs)**, 30, 153
- service sshd commands**, 282–283
- services edge**, 30–31
- Session Manager Daemon (SMD)**, 140, 143
- SFC (service function chaining)**, 64
- SGW (Service Gateway)**, 64
- shared networking**
 - Cisco IOS-XE configuration, 115–117
 - Cisco IOS-XR configuration, 117–122
 - global VRF route table*, 121
 - LXC spec file*, 122

- TPA configuration*, 121
- TPNNS (third-party network namespace)*, 119–121, 122
- XRNNS (XR network namespace)*, 117–119
- Cisco Nexus OS configuration, 122–125
 - default network namespace*, 122, 124
 - Guest Shell*, 124–125
 - management network namespace*, 122–124
- on IOS-XR platforms, 216
- output, 107–108
- overview of, 106–107
- support matrix for, 125
- shells. *See* Bash; Guest Shell
- show bash-shell command, 256–257
- show guestshell command, 243–245
- show iox-service command, 82
- show ip arp command, 294–295
- show kim status command, 195
- show mac address-table command, 294
- show NETCONF-yang sessions command, 329–330
- show platform command, 364
- show processes kim command, 194–195
- show system internal virtual-service event-history debug command, 279
- show version command, 364
- show virtual-service detail command, 278–279
- show virtual-service list command, 277–278, 279
- Simple Network Management Protocol (SNMP), 327, 330
- simplicity, 5
- SIP (SPA Interface Processor), 144
- Small Computer System Interface over IP (iSCSI), 239
- Smart Licensing, 186
- SMD (Session Manager Daemon), 140, 143
- SNMP (Simple Network Management Protocol), 327, 330
- SOA (Service Oriented Architecture), 13
- SOA (Start of Authority), 371
- sockets, verification of, 282
- software as a service (SaaS), 20
- software stability check, 362
- Software-Defined Access (SDA), 55
- software-defined networking. *See* SDN (software-defined networking)
- Solenoid, 406
- SPA Interface Processor (SIP), 144
- Spanning Tree Protocol (STP), 27, 237, 395
- Spec files, IP_Move application, 295
- SRv6 (Segment Routing v6), 420–421
- SSH (Secure Shell), 395
 - access for OAC (Open Agent Containers), 282
 - accessing Bash with, 258–260
 - accessing Guest Shell with, 245–248
 - activating, 246–247
 - NETCONF Agent and, 334
 - TCP port configuration for, 281
- SSIDs (Service Set Identifiers), 30, 153
- standalone fabric, 239
- standard error descriptor, 367
- Start of Authority (SOA), 371
- startup-config command, 395
- stderr, 367
- stopping Docker containers, 267–268
- storage
 - IOS-XE, 153–154
 - IOS-XR, 198–199
- storage area network (SANs), 12
- storage virtualization, 12–13
- STP (Spanning Tree Protocol), 27, 237, 395

sudo virsh command, 74

switches

- Catalyst. *See* Catalyst switches
- IES (Industrial Ethernet Switches), 55
- network device virtualization, 26–29
- network infrastructure virtualization, 23–27
- virtual, 414–415
- VSL (Virtual Switch Link), 27
- VSS (Virtual Switching System), 27–29

switching platforms, 142–144

switchover, container persistence through, 266–267

syncfd, 327, 328

synchronization, NX-OS and Guest Shell, 252

systemctl commands, 169, 246–247

systemd, 246

T

tables

- ARP (Address Resolution Protocol), 251, 252–253
- global VRF route, 121
- RIB (Routing Information Base), 406

TACACS+ protocol

- NETCONF Agent, 327
- RESTCONF Agent, 320–321

.tar files, importing, 172

TCP port configuration, 281

tcpdump, 407–408

tenants, multitenancy, 19–20

testing, infrastructure for, 363

third-party applications, 405–406

- Cisco Kinetic, 408
- DNS (Domain Name System), 409
- NetBeez Agent, 409–410
- OWAMP (One-Way Active Measurement Protocol), 407
- perfSONAR, 408–409

Solenoid, 406

tcpdump, 407–408

TWAMP (Two-Way Active Measurement Protocol), 407

third-party hosting plane, 193

third-party network namespace (TPNNS), 119–121, 122, 195, 212–213, 224–225

TLS (Transport Layer Security), 34, 321

toolchain, Docker, 175

tool/library installation (IOS-XE), 168–169

tools, 305–318

top talkers, 363

touch /var/lib/dhcp/dhcpd.leases command, 366

TPA configuration, 121, 217, 224

TPNNS (third-party network namespace), 119, 121–122, 195, 212–213, 224–225

traffic

- capture of, 394–395
- classification of, 395–396
- expected versus unexpected, 391
- profiling, 363

transistors, 2

Transport component (NX-API), 306

Transport Layer Security (TLS), 34, 321

transport layer socket sync, 193

Trunking protocols, 24–25

tshark command, 394–395

Turing, Alan, 1–2

Turing Machine, 2

TWAMP (Two-Way Active Measurement Protocol), 407

U

Ubuntu server, LXC (Linux containers) in, 84

UCS (Unified Computing System), 10, 412

- Ultra Services Platform, 53–54, 415–416
 - unexpected traffic, 391
 - Unicast Reverse Path Forwarding (uRPF), 239
 - Unified Computing System (UCS), 10, 412
 - Unified Fabric, 239
 - unmanaged switches, 24
 - unnumbered routing, 128–129
 - UPF (User Plane Function), 416
 - upgrading OAC (Open Agent Containers), 284–285
 - uRPF (Unicast Reverse Path Forwarding), 239
 - use cases, 361
 - anomaly detector application, 391–398
 - floodlight application*, 392–396
 - high-level procedure for*, 392
 - objectives of*, 391
 - configuration consistency check, 362
 - control plane health check, 362
 - DHCP Docker container deployment, 363–369
 - Catalyst switch configuration*, 363–364
 - container creation*, 364–368
 - container installation in Catalyst 9000*, 368–369
 - DNS Docker container deployment, 369–375
 - container creation*, 373–374
 - container installation in Catalyst 9000*, 374–375
 - preparation for*, 370–373
 - HAProxy and Node containers, 375–384
 - Docker image creation*, 377–378, 381–382
 - HAProxy Docker container installation*, 382–384
 - HAProxy load balancer setup*, 380–381
 - project initiation*, 375–376
 - Web server configuration*, 376–377
 - Web server Docker container installation*, 378–380
 - hardware/software stability check, 362
 - infrastructure for proof-of-concept and testing purposes, 363
 - inventory management, 362
 - NX-OS Docker health check, 398–404
 - config.ini file*, 399
 - Control Plane Policing (CoPP) counters*, 399–401
 - docker-compose.yml file*, 399
 - high-level procedure for*, 392–398
 - objectives of*, 398
 - running*, 401–404
 - operational data monitoring, 363
 - resource usage and scalability check, 362
 - S-BFD (Seamless BFD) for rapid failure detection, 384–391
 - client, hosting on server*, 390–391
 - discriminators*, 386
 - Docker images*, 388
 - overview of*, 385
 - reflector sessions*, 386–387
 - reflectorbase, hosting on XR devices*, 388–389
 - as VNF (virtual network functions)*, 387–388
 - traffic profiling and top talkers, 363
 - User Plane Function (UPF), 416
 - users, segmentation of, 25–27
- ## V
-
- /var/lib/docker file*, 263
 - /var/lib/lxd/containers/ directory*, 211
 - vCPU (virtual CPU), 9
 - VDCs (virtual device contexts), 29, 239

- Vector Packet Processor (VPP), 416, 420
- vehicles, connected, 53–54
- veth interface, 100–101, 102–104
- VIC (virtual interface cards), 412
- VIRL (Virtual Internet Routing Lab), 7
- virsh command, 74, 213–214, 215, 232–234
- Virtual Arbor Edge Defense, 418
- virtual CPU (vCPU), 9
- virtual device contexts (VDCs), 29, 239
- virtual interface cards (VIC), 412
- Virtual Internet Routing Lab (VIRL), 7
- Virtual Machine Disk (VMDK), 15
- virtual machine file systems, 169–170
- Virtual Machine Monitor (VMM), 7
- virtual machine-based applications, 166–175
 - components of, 151
 - installing and running, 172–175
 - IOx package creation with YAML, 170–172
 - network configuration, 174
 - tool/library installation, 168–169
 - virtual machine file systems, 169–170
 - virtual machines, building, 167–168
- virtual machines, 14–15
- virtual network functions. *See* VNFs (virtual network functions)
- virtual network interface cards (vNICs), 15, 155–156
- Virtual PortChannel (vPC), 28, 395
- Virtual Route Forwarding (VRF), 30–31
- Virtual Router Redundancy Protocol (VRRP), 395
- Virtual Switch Link Protocol (VSLP), 27
- Virtual Switch Link (VSL), 27
- Virtual Switching System (VSS), 27–29
- virtual user plane forwarder (vUPF), 64
- Virtual Wide Area Application Service (vWAAS), 67
- virtualization
 - architecture and components of, 6–8
 - cloud computing versus, 13
 - containers, 15–17
 - control plane, 33–34
 - definition of, 6
 - design considerations
 - high availability*, 18
 - multitenancy*, 19–20
 - workload distribution*, 19
 - full, 6–7
 - hardware-assisted, 7
 - history of, 2–3, 23–29
 - enterprise virtualization*, 30–31
 - network device virtualization*, 26–29
 - network infrastructure virtualization*, 23–27
 - service providers*, 31
 - software-defined networking*, 32–33
- IBN (Intent-Based Networking), 57
- IoT (Internet of Things), 51–57
 - Cisco Kinetic*, 52–53
 - Cisco Ultra Services Platform*, 53–54
 - devices in*, 51–57
 - manufacturing use case*, 55–57
- motivation and business drivers for, 3–6
 - cost optimization*, 5
 - resilience*, 5
 - resource optimization*, 4–5
 - simplicity*, 5
- nested, 7
- network, 10–11

- NFV (network functions virtualization)
 - elements in*, 48
 - modern network design with*, 47
 - virtual network service deployment*, 48
 - paravirtualization, 7
 - routers and switches, 414–415
 - SDN (software-defined networking), 33, 47
 - APIs (application programming interfaces)*, 36–37
 - control plane virtualization*, 33–34
 - programmability*, 38–42
 - SDN controllers*, 34–36
 - server, 8–10
 - serverless computing, 17–18, 421–423
 - storage, 12–13
 - VNFs (virtual network functions), 405, 411
 - containers and service chaining*, 418–421
 - definition of*, 48
 - S-BFD (Seamless BFD) as*, 387–388
 - Virtualization Manager (VMAN), 67, 146–148
 - VirtualPortGroup, 125–126
 - configuration, 127
 - creating, 158, 172
 - IOS-XE requirements, 154–155
 - virtualportgroup command, 154
 - virtual-service command, 74, 280
 - VLAN Trunking protocol, 24–25
 - VLANs (virtual LANs), 23–27
 - VMAN (Virtualization Manager), 67, 146–148
 - VMDK (Virtual Machine Disk), 15
 - VMM (Virtual Machine Monitor), 7
 - VMware, 30, 170
 - VNFs (virtual network functions), 405, 411
 - containers and service chaining, 418–421
 - NSH (Network Service Header)*, 419–420
 - SRv6 (Segment Routing v6)*, 420–421
 - definition of, 48
 - S-BFD (Seamless BFD) as, 387–388
 - vNICs (virtual network interface cards), 15, 155–156
 - vPC (Virtual PortChannel), 28, 395
 - VPP (Vector Packet Processor), 416, 420
 - VRF (Virtual Route Forwarding), 30–31
 - VRF namespace, application hosting in, 226–232
 - with LXC (Linux containers)*, 229–232
 - namespace configuration*, 226–229
 - VRF sync, 193
 - VRF-Lite, 25
 - VRRP (Virtual Router Redundancy Protocol), 395
 - VSL (Virtual Switch Link), 27
 - VSLP (Virtual Switch Link Protocol), 27
 - VSS (Virtual Switching System), 27–29
 - vThunder Secure Application Services, 418
 - vtysserverutild, 328
 - vUPF (virtual user plane forwarder), 64
 - vWAAS (Virtual Wide Area Application Service), 67
- ## W
-
- WAPs (wireless access points), 143
 - WCM (Wireless Client Manager), 140, 143

Web servers

configuration

*for Docker containers, 378–380**for HAProxy and Node containers, 376–377*

S-BFD clients, hosting, 390–391

/well-known/host-meta resource, 321

Wi-Fi 6 technologies, 53

wireless access points (WAPs), 143

Wireless Client Manager (WCM), 140, 143

wireless networking

5G, 53

Wi-Fi 6 technologies, 53

worker nodes (Kubernetes), 80–81, 93–94

workload distribution, 19

WRL environment, 207

X

XaaS (Anything as a Service), 17–18, 20

Xen, 3

XML (Extensible Markup Language), 38, 306

XR control plane, 193

XRNNS (XR network namespace), 117–119, 195

Y-Z

YAML (YAML Ain't Markup Language), 38

Alpine Linux container deployment, 94

Cisco Application Hosting Framework (CAF), 69

Cisco Service Container descriptor file, 67–68

interface data model, 40–42

IOx package creation with, 170–172

package creation with, 162–165, 180–181

*package.tar file, 163–165**package.yaml descriptor file, 163*

YANG (Yet Another Next Generation), 39–42

yum install command, 70, 204, 288

yum repolist command, 203, 288

YUM repositories

configuration, 203

default, 203

installing native applications from, 204

Zend Studio, 339