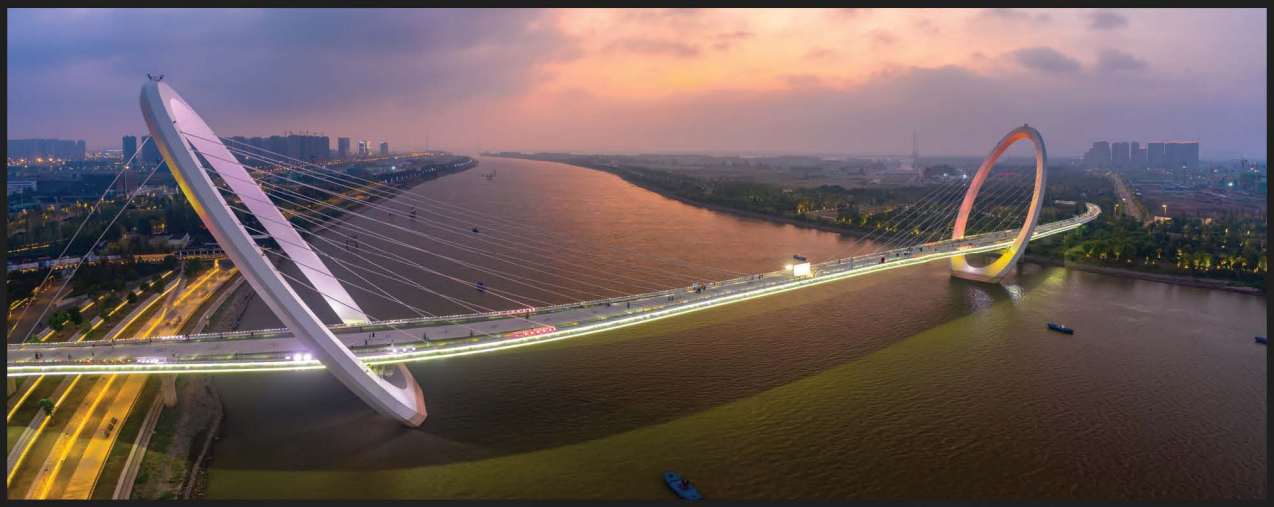


3RD EDITION

Mastering OpenStack

Implement the latest techniques for designing and deploying an operational, production-ready private cloud

OMAR KHEDHER

Mastering OpenStack

Implement the latest techniques for designing and deploying an operational, production-ready private cloud

Omar Khedher

‹packt›

Mastering OpenStack

Copyright © 2024 Packt Publishing

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

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

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

Group Product Manager: Preet Ahuja
Publishing Product Manager: Suwarna Rajput
Book Project Manager: Srinidhi Ram
Senior Editor: Mohd Hammad
Technical Editor: Rajat Sharma
Copy Editor: Safis Editing
Language Support Editor: Safis Editing
Proofreader: Mohd Hammad
Indexer: Pratik Shirodkar
Production Designer: Alishon Mendonca
DevRel Marketing Coordinator: Rohan Dobhal

First published: July 2015
Second edition: April 2017
Third edition: November 2024

Production reference: 1181024

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK

ISBN 978-1-83546-891-3

www.packtpub.com

Contributors

About the author

Omar Khedher is a cloud architect with more than 12 years of experience in the cloud computing market. He has been involved in several private and public cloud projects, including OpenStack, AWS, and Azure. He has also authored several other books, including the first and second editions of *Mastering OpenStack*, *OpenStack Sahara Essentials*, and *Extending OpenStack*. Omar has a few academic publications about cloud performance. As a part of the *Cloudreach* team, an international cloud consulting company, he helps customers to design, advises on, and leads projects targeting cloud migration and innovation.

I want to thank the special people who have supported me on this long journey of writing the third edition, including my wonderful parents and brothers, my lovely cats, my princess for her patience, and, of course, my truly amazing Cloudreachers at Eviden. Thanks to Belgacem, Andre, Miko, and Cristian for always being around for support. I also thank my music fellows, who were a source of inspiration during the jam breaks.

The Packt team has been, as always, a great source of support for me during the writing process – a huge round of applause to them.

About the reviewers

Innocent Twagirimana is a senior cloud/DevOps engineer with a total of 12 years of experience in information technology, including seven years specifically in cloud and DevOps engineering. He holds a master's degree in information technology from Carnegie Mellon University and certifications such as **Certified OpenStack Administrator** (**COA**), **Certified Kubernetes Administrator** (**CKA**), and **Linux Foundation Certified System Administrator** (**LFCS**). Innocent has successfully deployed and operated production OpenStack and Kubernetes clusters. Passionate about contributing to open source, he also enjoys exercising and playing basketball in his spare time. He provides freelance services to IoT-based product companies, writing technical reviews and projects, as well as providing consultancy to organizations.

I would like to extend my deepest gratitude to my family, whose unwavering support and understanding allowed me to dedicate countless nights and weekends to reviewing this book. Their patience and encouragement have been invaluable, and I could not have completed this work without their love and support.

Mohankumar is an accomplished OpenStack network engineer with extensive experience in designing and managing sophisticated cloud network solutions. Holding a master's degree in information technology from Anna University, he has consistently delivered numerous projects, implementing scalable and secure networking architectures tailored to diverse business needs.

Since early 2014, Mohankumar has been actively involved with the OpenStack community, sharing insights through published articles and speaking engagements. As a reviewer, he leverages his extensive experience to provide valuable, in-depth analysis and feedback on technical publications, helping to advance the industry's body of knowledge.

I would like to express my deepest gratitude to my family for their unwavering support and encouragement. To my wonderful spouse and my two amazing kids, your love and patience have been my greatest source of strength. A special thanks to the author for the incredible work and to the publisher for the opportunity to contribute to this publication.

Geetika Batra is an R&D engineer at Broadcom with 8+ years of experience in software engineering, specializing in Kubernetes and cloud-native technologies. She has contributed significantly to GPU PCIe support and vGPU implementation in the Cluster API for vSphere. A certified Kubernetes administrator, Geetika holds a B. Tech in electronics and communication. Her career spans VMware, Broadcom, Oerlikon, and Red Hat, where she's worked on both open source and proprietary projects.

Thanks to my family, friends, and the Kubernetes and DevOps community for their support in this ever-changing field. Gratitude to the trailblazers who keep our work exciting, and to the author and publisher for the opportunity to review this book. I appreciate everyone who has worked on making Mastering OpenStack a dynamic and rewarding endeavor.!

Table of Contents

Preface xiii

Part 1: Architecting the OpenStack Ecosystem

1

Revisiting OpenStack – Design Considerations 3

OpenStack – a plethora of innovations 5
Building blocks Of OpenStack – the control plane 6
Keystone – the authentication and authorization service 7
Nova – the compute service 8
Placement – the scheduling service 8
Glance – the imaging service 9
Swift – the object storage service 10
Cinder – the block storage service 11
Manila – the shared filesystems service 11
Neutron – the networking service 11
Ceilometer – the telemetry service 12
Aodh – the alerting service 13
Horizon – the dashboard service 13
Non-OpenStack services 14
Constructing the motivation 16
Application development accelerator 16
Cloud application enabler 17
HPC supporter 18
Network functions virtualization moderator 18
Big data facilitator 18
Private cloud service provider 19
Public cloud service provider 19
Picking up the pieces 19
Architecting the cloud 26
Drafting the conceptual design 27
Drafting the logical design 28
Drafting the physical design 33
Summary 40

2

Kicking Off the OpenStack Setup – The Right Way (DevSecOps) 43

Removing silos – DevOps 44
Shifting to the left – DevSecOps 45
Securing the cloud – moving with DevSecOps 46
DevSecOps and OpenStack 47

Coding the cloud 47
Deploying in the cloud 48
Ansible in a nutshell 48
Containerizing the cloud 51
Building the picture 53
Setting up a development environment 53
Preparing the local environment 54
Running the local cloud 56
Extending the deployment 62
Installing the deployer 63
Actioning the deployer 66
Deploying with security 72
Summary 77

3

OpenStack Control Plane – Shared Services 79

The OpenStack control plane 80
Running the control plane – cloud controller 80
Shared services – infrastructure services 92
Arming the control plane – prepare for failure 93
Planning for more 95
Extending the deployment 96
Initializing the network 97
Listing the cluster specs 99
Preparing the infrastructure code 99
Configuring the host groups 100
Parameterizing the environment 107
Advanced configuration 108
Deploying the environment 109
Summary 113

4

OpenStack Compute – Compute Capacity and Flavors 115

The compute service components 116
API 117
Scheduler 117
Conductor 117
Compute 117
Console access 118
Reasoning for the hypervisors 118
Compute segregation 120
Infrastructure segregation 120
Workload segregation 127
Pre-filtering 128
Resource provider allocation 128
Filtering 131
Weighting 132
Extending compute for containers 136
Magnum 136
Zun 139
Expanding the compute farm 142
Summary 144

5

OpenStack Storage – Block, Object, and File Shares 145

Defining Block Storage – Cinder 146
Expanding storage with multiple backends 147
Storage filtering and scheduling 154
Revisiting object storage – Swift 157
Revisiting Swift 157
Building a Swift cluster 158
Running Swift 160
Exploring file share services – Manila 164
Manila under the hood 165
Running Manila 166
Summary 169

6

OpenStack Networking – Connectivity and Managed Service Options 171

Exploring Neutron's core components 172
Demystifying Neutron agents 173
Network categories 174
The core of networking – Neutron plugins 174
Core plugin – Modular Layer 2 174
Building virtual switching 176
Configuring cloud routing 188
Routing tenant traffic 188
Neutron dynamic routing 195
Joining more networking services 197
Load balancer as a service 197
Summary 202

Part 2: Operating the OpenStack Cloud Environment

7

Running a Highly Available Cloud – Meeting the SLA 207

Exploring HA strategies 208
Measuring HA 208
Designing for HA 210
Preparing for HA 211
Designing with load balancing 211
HA for the database 213

HA for the message bus 217
Deploying for HA 218
HA for networking 222
Managing instance failover 230
Deploying Masakari 232
Summary 234

8

Monitoring and Logging – Remediating Proactively 237

OpenStack infrastructure monitoring 238
Prometheus in a nutshell 238
Deploying infrastructure monitoring 239
Exploring the monitoring data 243
OpenStack workload monitoring 250
Telemetry and charging back 251
The Ceilometer anatomy 252
The Gnocchi data store 254
Alerting with Aodh 255
Deploying the telemetry service 257
Grasping centralized logging 259
OpenSearch under the hood 260
Deploying OpenSearch 261
Summary 268

9

Benchmarking the Infrastructure – Evaluating Resource Capacity and Optimization 269

Empowering the database 270
Running with caching 270
Deploying caching 272
Benchmarking the cloud 276
Rally in action 276
Profiling the cloud 287
Profiler in action 287
Watching the cloud 289
Watcher in action 290
Summary 298

Part 3: Extending the OpenStack Cloud

10

OpenStack Hybrid Cloud – Design Patterns 301

Hybrid cloud under scope 302
Public cloud strategy 302
Hybrid cloud strategy 306
OpenStack meeting the public world 309

Enabling the hybrid cloud 311
Cloud bursting 312
Security in the hybrid cloud 313
Governance first 314
Network security 315
Hardening operation security 316
Summary 317

11

A Hybrid Cloud Hyperscale Use Case – Scaling a Kubernetes Workload 319

Qualifying a hybrid architecture 320
Adopting a cloud-agnostic stack 320
Preparing for a hybrid model 322
Designing a containerized hybrid cloud 324
Kubernetes in a nutshell 325
Designing a decentralized models 327
Designing a centralized model 331
Summary 353

Index 355

Other Books You May Enjoy 368

Preface

Welcome to *Mastering OpenStack*, the third edition of the *Mastering* series, which will provide you with the latest updates and the state of the art in deploying and running private cloud infrastructure based on OpenStack. The adoption of OpenStack has often been challenging for organizations due to the complexity of its ecosystem and fast development across different releases. This book will address those challenges by taking an iterative approach for each core component and preparing for a production-ready private cloud design. In each chapter, the book draws inspiration from the most commonly applied best practices for deploying and managing the infrastructure. This includes leveraging infrastructure as code, CI/CD, and DevSecOps paradigms. As the book will cover new OpenStack services and advanced concepts, automation will be your *guardian angel* to successfully deploy them. This book is also suitable for new OpenStack users as it will iterate through its core components and deploying them using containers. Extensive knowledge of the containerization technology is not required but a basic background is recommended. Most importantly, the book focuses on production quality and the most popular deployments inspired by a wide range of industries. The OpenStack world can present a massive number of options for services and deployments. The book is intended to help you understand how to architect and design a private cloud out of the box to meet your organization's needs. Toward the end of this edition, the book will open the curtains on a recent trend in the cloud market: the hybrid cloud. OpenStack takes the role of the private cloud and joins with an AWS environment to build a hybrid cloud setup. At the end of the book, more opportunities will be explored for running large-scale environments across different cloud providers where OpenStack fits well as a private cloud solution.

Who this book is for

This book is for cloud architects and administrators who have a basic understanding of the OpenStack ecosystem as well as the cloud and virtualization concepts. The book is useful for enterprise consultants and cloud developers who have working experience in the public cloud. Familiarity with Linux command lines and basic knowledge of containerization and network topologies are recommended to successfully enjoy most of the content of the book.

What this book covers

Chapter 1, *Revisiting OpenStack – Design Considerations*, summarizes the latest features of the main architectural services of OpenStack. The chapter introduces the latest updates of each core component of Antelope and later releases. Initial logical and physical design models will be drafted and used as an architecture reference for later chapters.

Chapter 2, *Kicking Off the OpenStack Setup – The Right Way (DevSecOps)*, starts by covering the concept of DevOps concept with the merge of security aspect. The chapter explores how DevSecOps helps to manage, deploy, and automate a large OpenStack private cloud infrastructure in the most agile and secure ways. The chapter introduces the Kolla Ansible project, which will be used to deploy different OpenStack cloud infrastructure pieces based on containers.

Chapter 3, *OpenStack Control Plane – Shared Services*, zooms in on the common OpenStack services designed to run in cloud controller nodes. The chapter dives into the latest OpenStack updates introduced into its control plane, including core service APIs, messaging, and database services. Based on the deployment based on containers, the chapter explores in more depth different pieces of the Kolla Ansible repository, targeting the OpenStack control plane.

Chapter 4, *OpenStack Compute – Compute Capacity and Flavors*, revisits the OpenStack compute Nova service in more depth. Scheduling and the Placement service will be covered, including advanced ways to carry out resource allocation to run instances in compute nodes. The chapter explores the latest updates to compute design for large-scale OpenStack deployments by covering the concepts of availability zones, host aggregation, and cells. A good part of the chapter is dedicated to learning about incubated projects targeting containerization technology in OpenStack. Zun and Magnum will be highlighted and deployed the same way as other services using infrastructure as code.

Chapter 5, *OpenStack Storage – Block, Object, and File Shares*, explores different storage options within the latest OpenStack releases, starting from the Antelope release. The chapter reveals different backend integrations with the Cinder block storage service, such as NFS and Ceph. More updates on block storage scheduling are highlighted in more detail. A succinct update overview for Manila and Swift will be given. The covered storage services will join the deployment model using the Kolla Ansible repository.

Chapter 6, *OpenStack Networking – Connectivity and Managed Service Options*, explains in more depth the Neutron networking service in OpenStack. The chapter introduces the reworked layout of the network architecture for large-scale deployment. Neutron plugins will be explored with the latest OpenStack release updates, including Open vSwitch and OVN. Routing is discussed and we show how it works under the hood by leveraging the Neutron plugin. Additional services, such as the new version of Load Balancing as a Service, codenamed Octavia, is demonstrated throughout the chapter.

Chapter 7, *Running a Highly Available Cloud – Meeting the SLA*, covers different techniques and design patterns to increase availability and scalability in each OpenStack control plane layer. The chapter explores different ways to achieve routing redundancy specifically in Neutron. It also goes beyond the infrastructure layer and exclusively introduces an automated approach to manage instance availability using a new trending OpenStack service codenamed Masakari.

Chapter 8, *Monitoring and Logging – Remediating Proactively*, illustrates a consolidated solution to run a sophisticated monitoring solution for a large OpenStack cloud setup. The chapter introduces Prometheus as a single pane of glass for all the OpenStack monitoring metrics. It also explores a simple way of visualizing and centralizing metrics in one system using Grafana. Succinct updates on the Ceilometer telemetry service in OpenStack are covered. The chapter unleashes a powerful and automated way to digest and visualize large amounts of OpenStack logs using OpenSearch.

Chapter 9, *Benchmarking the Infrastructure – Evaluating Resource Capacity and Optimization*, goes through more advanced operational tasks targeting OpenStack performance and resource optimization. The chapter discusses how to evaluate your cloud infrastructure core services and limits by performing benchmarking exercises using the Rally tool. It also takes the performance dilemma to the next level by enabling ways of tracing different OpenStack calls using a dedicated tool developed for OpenStack codenamed OSProfiler. The chapter highlights a great addition to the OpenStack project for resource optimization codenamed Watcher.

Chapter 10, *OpenStack Hybrid Cloud – Design Patterns*, suggests new ways of implementing, deploying, and integrating OpenStack beyond the private cloud model. The chapter discusses the rising trend of adopting a hybrid cloud model for several use cases. Hybrid cloud design patterns are discussed throughout the chapter and we explore the marriage between the public and private clouds based on OpenStack.

Chapter 11, *A Hybrid Cloud Hyperscale Use Case – Scaling a Kubernetes Workload*, wraps up this edition with exclusive content on scaling workloads between a private OpenStack cloud and an AWS public cloud environment. The chapter crystallizes the concept of duo containers and microservices to achieve a hybrid cloud model. A workload based on Kubernetes will be demonstrated to run between private and public cloud worlds and we cover a trending Canonical tool code named Juju to federate and manage workloads across clouds.

To get the most out of this book

Software/hardware covered in the book	**Operating system/hardware requirements**
OpenStack (Antelope or later releases), AWS account Python, YAML, JSON Ansible, Kolla, Jenkins, Docker, Kubernetes Ceph, Canonical Juju, KubeFed	Linux (Ubuntu 22.04), Vagrant 2.4.1 or later Test Environment (1 Host) • 2 CPU or more • 8 GB RAM or more • 50 GB Disk space or more Minimal Dev/Prod Environment (2 Machines minimum) per Host: • 2 CPUs • 8 GB RAM • 100 GB Disk space or more

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Mastering-OpenStack-Third-Edition`. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

> **Important Note**
> From the inception to the publishing of this book, there have been a couple of major version changes in OpenStack. Due to this, the code mentioned in some sections of the book may not work without some adjustments. The instructions for any necessary setup or troubleshooting steps, as well as updated code files, will be included for each chapter in the above mentioned GitHub repository.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Setting `enable_magnum` to `yes` in the `/kolla-ansible/etc/kolla/globals.yml` file will enable the Magnum API service."

A block of code is set as follows:

```
…
[magnum:children]
control
[magnum-api:children]
magnum
[magnum-conductor:children]
magnum
…
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
"provider_summaries": {
        "ad54422c-b345-5v44-ba23-00cad76e376a": {
            "resources": {
                "DISK_GB": {
                    "used": 0,
```

```
                    "capacity": 2000
                }
            },
            "traits": ["SRG_SHARES_STORE"],
```

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

```
# openstack allocation candidate list --resource VCPU=4,
MEMORY_MB=2048,DISK_GB=500GB --required   HW_CPU_X86_SVM
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "On the Jenkins dashboard, configure the installed Anchore plugin by pointing to **Manage Jenkins** and clicking on **Configure System**."

> **Tips or important notes**
> Appear like this.

Get in touch

Feedback from you is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata` and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Share Your Thoughts

Once you've read *Mastering OpenStack*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



`https://packt.link/free-ebook/9781835468913`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1: Architecting the OpenStack Ecosystem

The first part of the book will go through the latest updates of OpenStack, including Antelope and subsequent releases. This will highlight any new additions and required changes to prepare for the deployment of a private cloud setup. This part will review all the required core components to design a robust layout and implement it with the most stable operational capabilities, which will include all the required services with different configuration options for both the control and data plane. This first part of the book is an essential stage for the next parts, as it will tackle the ways of deploying different OpenStack services through Infrastructure as Code and DevSecOps styles.

This part has the following chapters:

- *Chapter 1*, *Revisiting OpenStack – Design Considerations*
- *Chapter 2*, *Kicking Off the OpenStack Setup – The Right Way (DevSecOps)*
- *Chapter 3*, *OpenStack Control Plane – Shared Services*
- *Chapter 4*, *OpenStack Compute – Compute Capacity and Flavors*
- *Chapter 5*, *OpenStack Storage – Block, Object, and File Shares*
- *Chapter 6*, *OpenStack Networking – Connectivity and Managed Service Options*

1

Revisiting OpenStack – Design Considerations

"I have found you have got to look back at the old things and see them in a new light."

- John Coltrane

Since the last edition of the *Mastering OpenStack* series, the OpenStack community has kept the momentum by growing and innovating around its ecosystem. At the time of writing this edition, the OpenStack design has been going through different cycles of improvements by including new projects and providing seamless integration with existing systems to respond to organization demands and custom features. Since the first *Austin* release back in 2010, the innovation kept opening new opportunities for companies and quickly adopting a stable private cloud setup to stay ahead in the market. The challenges of handling a scalable infrastructure have been felt by big and medium players who joined the OpenStack evolvement. Today, as the first page of this edition is being written, 14 years and 28 releases have already passed through different OpenStack releases, from *Austin* back in the day to the latest *Dalmatian* release. That is a full circle back to the beginning of the alphabet. A lot of experiences and insights have been revealed that have boosted the OpenStack community and made it the world's leading open source cloud platform.

Numbers show how successfully OpenStack has been adopted by mid to large enterprises. As per the annual *OpenStack User Survey* results in the OpenStack blog in November 2022, it was declared that more than 40 million cores are running in production across 300 deployments empowered by OpenStack.

The OpenInfra community has been committing enormously to hardening the most stable versions of OpenStack software. Starting from the *Antelope* release that was launched in March 2023 and, more recently, the *Dalmatian* release in October 2024, organizations have quickly upgraded, and the good news is a more stable and reliable version than ever. The secret sauce of this great achievement is that the community has switched gears on stabilizing basic OpenStack services, including compute,

storage, and network services, increasing the cadence of release testing of each service extensively and thus leaving neither possible gaps nor potential vulnerabilities that could raise bugs in deployments in production. On the other side of the story, by going through the map of each OpenStack release, you might notice some extension projects appearing and disappearing. Community contributors did the right thing, discounting non-stable features that would hinder the maturity of the OpenStack private setup. Today, the release is exposing basic services in addition to more projects and features in a stable upstream. The other bright side of the story is that the OpenStack ecosystem has always been one step ahead to adopt new trends of market technologies that include containerization, serverless, big data analysis, and so on.

Today, either starting a new OpenStack adventure or turning a new page for a new update would bring great cost savings, flexibility in IT handling compared to old-school virtualization alternatives, and an efficient solution to hyperscale with minimum concerns.

OpenStack's state-of-the-art documentation and resources can be found more abundantly than ever on the internet. There are plenty of different options to design and deploy a complete OpenStack environment. On the other hand, the paradox of choices between different ways of designing and running a full ecosystem can be challenging even for experienced administrators, architects, and developers. That could be phrased differently as *too much can be too little*! The adoption of OpenStack can be a struggle and has always been a challenge, from design phases to deployments and operational days. As stated previously, you can think of many reasons for this, but mainly, being such a versatile software can make it overwhelming!

Throughout this book, we will go through the updated pieces of the OpenStack ecosystem and divide and conquer together with a step-by-step approach to design, deploy, and operate a scalable OpenStack environment that would respond to your needs and requirements.

To address the challenges of the OpenStack ecosystem complexity and allow newcomers to enjoy this cloud journey, we will define a state vision for a private cloud adoption strategy throughout the first chapter. Without a well-crafted basic knowledge of this ecosystem, it would be more difficult to make later decisions on empowering running clusters in production when production days start knocking on the door. A very attractive citation from Robert Waterman, an expert on business management practices, is *"A strategy is necessary because the future is unpredictable."*

When it comes to a complex system such as the OpenStack ecosystem, setting up the right resources is required as we cannot predict with 100% accuracy. OpenStack is designed with much more flexibility and a loosely coupled architecture. This way, it is up to us to use those key elements to make capacity decisions considering short- and long-term goals for growth and availability and respond to new workloads based on existing resources that we aim to apply in the near and long term.

Our OpenStack journey will continue in this edition by covering the following topics in this chapter:

- Revisiting and highlighting the latest updates of the OpenStack core ecosystem
- Demystifying the logical architecture based on the latest releases introduced from the *Antelope* release
- Drafting a first physical design to ensure a seamless deployment in later stages
- Preparing for a large-scale OpenStack environment through capacity planning as part of our cloud strategy

OpenStack – a plethora of innovations

It might be confusing to understand the state of the art of innovation when it comes to OpenStack software offerings. The rise of this open source project has been unique since its birth 14 years ago. Through its different releases, the OpenStack community has kept developing what a private cloud solution can offer based on ongoing technology trends. When the first releases of Kubernetes (container orchestration technology) were made public, subsequent OpenStack releases quickly dedicated a cycle to include services around the OpenStack ecosystem that would facilitate container management out of the box. One of the OpenStack success stories is undoubtedly its vision of staying one step ahead. There is a clear reason why OpenStack remains the fourth largest open source community worldwide: trust in its main core services. Since the first days of OpenStack, mid and large enterprises have invested and keep contributing to its software to boost its major core services. As we will see later, compute, network, identity, image, and storage services are counted as the most basic and core components of the OpenStack ecosystem. Within each new release, more enhancements and extensive development have been done to bring a more alive version of each of them. That brings more contributors to see services more alive, taking more workloads, addressing more demand, and unblocking enterprises to take advantage of a more agile and flexible infrastructure. More big players have joined the cloud era from the early days, such as IBM, Red Hat, HP, Rackspace, eBay, and more, to name but a few. Each of them associates different goals and projects from different expertise, such as bringing new features, plugins, bug fixing, and so on. Some of those contributors have built their own dedicated private cloud platform based on OpenStack and keep contributing to the open source world, making it a win-win deal.

Another key aspect of OpenStack novelty is the speed of extending its capabilities from a wider range of services similar to public major cloud providers such as **Amazon Web Services** (**AWS**), **Microsoft Azure**, and **Google Cloud Platform** (**GCP**). Starting from traditional **Infrastructure as a Service** (**IaaS**), the OpenStack ecosystem has tailored the next levels by bringing managed services and offering **Platform as a Service** (**PaaS**) as well as **Software as a Service** (**SaaS**) models. By skimming the surface of the speed of OpenStack growth, we might find out that the core services have reached a level of maturity that could unlock gazillions of doors of innovative features and solutions. On top of those services, more PaaS environments can be offered, such as databases, big data, container services, and so on.

The other side of the success of the OpenStack growth is the nature of the software itself. Written mostly in Python, since the first days of its development, the OpenStack software has been designed to offer a rich **application programming interface** (**API**). This was a game changer by enabling automation in almost everything and everywhere. Hence, each contributor can use the domain of expertise and seamlessly integrate new features into the software ecosystem, thanks to the nature of API design.

OpenStack's APIs have been of considerable significance in adopting a hybrid cloud approach. As we will see in *Chapter 11*, *A Hybrid Cloud Hyperscale Use Case – Scaling a Kubernetes Workload*, OpenStack offers the **Elastic Compute Cloud** (**EC2**) API, a compatible API to interact with the AWS world.

The innovation and continuous growth of the OpenStack ecosystem are without a doubt a success story thanks to the community's consistency and, most importantly, the development of OpenStack naturally with the modular design. Nowadays, it is possible to build an OpenStack private cloud for one or several purposes that can serve web applications for public hosting workloads, managed databases, big data, high-performance computing (HPC), or even several optimized containerized environments for rapid application development. Most importantly, getting acquainted with core and latest service updates is the must-have key to unlocking the cloud journey.

Building blocks Of OpenStack – the control plane

The OpenStack project has been launched to solve the IaaS paradigm. By adopting the pay-as-you-use model, underlying resources, including compute, networking, and storage, are exposed as pools and reserved on demand securely per the user's request. Throughout the development of the OpenStack ecosystem (considered in other literature as the *cloud operating system*), more open source projects have joined the emerging cloud software life cycle to extend its capabilities. As mentioned previously, the secret sauce of such project development comes from the natural API design, which consistently facilitates communication between services. The versatile number of services could sidetrack newcomers in understanding where and how fundamental parts of the ecosystem work to enable a well-designed architecture for custom needs and requirements. The following sections will iterate through what are considered the *core services* of OpenStack, to begin with. It is essential to get acquainted with each of these services as each OpenStack release, coming with new projects or services, relies on them. Being strategic in mastering core components and their capabilities will allow easier deployment of your future private cloud setup and, hence, extend its capabilities with less wasted time and effort. The following table shows the core services that will be tackled in the following sections:

Service	Code name	Description
Compute	Nova	Manages the virtual machine (VM) life cycle
Network	Neutron	Manages network environment per project
Identity	Keystone	Provides authentication and authorization information service

Service	Code name	Description
Block storage	Cinder	Manages the VM disks and snapshots life cycle
Object Storage	Swift	Accessible REST API storage for object data types such as images and media files
Image	Glance	Manages the VM image life cycle
Dashboard	Horizon	OpenStack frontend web interface
File sharing	Manila	Manages projects across shared filesystems
Scheduling	Placement	Helps to track provider's resource inventories and usage
Telemetry	Ceilometer	Provides data collection service for resource tracking and billing
Alarming	Aodh	Triggers actions and alarms based on collected metrics and configured rules

Table 1.1 – OpenStack core services

Important note

It is still possible to deploy an OpenStack environment without object storage and a dashboard. Some distributions come by default with both services enabled. For example, spinning up a VM does not necessarily require object storage. Additionally, the OpenStack **command-line interface** (**CLI**) supports all operations for each installation that can be directed without the OpenStack dashboard.

Keystone – the authentication and authorization service

The keywords *authentication* and *authorization* are the main functions of this identity service. **Keystone** is part of the control plane of the OpenStack ecosystem.

Security has been brought over from the earliest releases, and it became a necessity to dive further into the **AAA** (short for **Authentication, Authorization, Accounting**) service in OpenStack. Each request made to contact an OpenStack service must be validated by Keystone (via the identity API). The API response will return an authentication token to be used against the requested service (API call).

At this stage, the Keystone workflow might seem simpler than we thought. On the other hand, when we scale hundreds of hosts to fulfill thousands of requests in a short time, we should consider how to make sure that Keystone, as a critical service, is fully operational. That will be discussed in later chapters.

What has been brought from old OpenStack releases to the latest stable ones within the identity service is mainly the version of the API. Formerly, when installing the identity service, it was possible to keep running with version 2; starting from the *Grizzly* release, version 3 has been launched and can be installed. Today, the identity API version 2 has been deprecated in favor of newer releases. We will consider version 3 and above (at the writing of this book, version 3.14 is the latest within the *Ussuri* OpenStack release).

But what can an identity API offer? As mentioned previously, the design of all communications between OpenStack services is made via APIs (later, we will skim the surface for interaction even with non-OpenStack services via APIs). Consider that a failed simple authentication request might reflect a service failure from the end user's perspective. A good practice is to make sure, on the first iteration, that there are no issues with Keystone requests when troubleshooting a failing service request. Although the previous editions of this book did not dive deep into the Keystone workflow, we will revisit it in more detail in *Chapter 3*, *OpenStack Control Plane – Shared Services*.

Nova – the compute service

Austin, the first OpenStack release, brought the first version of the **Nova** project to spin up an instance and started dealing with compute management through a rich API. Unlike the aforementioned identity service, Nova has evolved through different releases to accommodate more amazing functions but at the cost of its complexity compared to other core services. The Nova service design has not changed a lot from the *Grizzly* release. On the other hand, it is still required to reiterate through the different Nova pieces that form the compute service engine in the OpenStack ecosystem.

> **Important note**
>
> There are plenty of capabilities brought by Nova in the latest OpenStack releases, which will be detailed in *Chapter 4*, *OpenStack Compute – Compute Capacity and Flavors*.

The `nova-api` service interacts with user API calls that manage compute instances. It communicates with other components of the compute service over a message bus.

The `nova-scheduler` service listens to the new instance request on the message bus. The job of this service is to select the best compute node for the new instance.

The `nova-compute` service is the process responsible for starting and terminating VMs. This service runs on the compute nodes and listens for new requests over the message bus.

The `nova-conductor` service handles database access calls from the compute nodes to limit the risk of database access by an attacker via a compromised host.

Placement – the scheduling service

Whenever it comes to a new system improvement or logical addition, the OpenStack community will create, move, or place different services or components to make that happen. One of the major updates

of the OpenStack ecosystem since the *Newton* release is the introduction of the **Placement** API service. Before the latter release, users struggled to identify the counter on different sets of resource providers, such as computation, networking, and storage allocated pools. That is where it is useful to dedicate a separate service, hence an API well connected with other services (such as Nova), to track those resource providers and keep an eye on their usage. The Placement service acts mainly as a resource inventory. Not only that, but the most pertinent addition has also made the process of filtering (via `nova-scheduler`) more fine-grained. In some other glossaries, you might find the word *prefiltering* as a reference for the new Placement service. This step is the preparation of available compute nodes by starting a filtering process before dealing with the scheduler based on some configurable specs and traits. The Placement service will be demonstrated in *Chapter 4*, *OpenStack Compute – Compute Capacity and Flavors*.

Glance – the imaging service

To launch an instance in OpenStack via the Nova service, an instance image will be required to successfully accomplish VM provisioning. **Glance** is one of the core services that deals not only with instance images (as a template source image) but also the snapshots that can be created from an instance. When it comes to the question of where Glance will store images and snapshots, the answer can be found in its latest supported driver configurations. As with many other OpenStack services, a number of storage options can be used to back the storage of images by Glance. Again, if we think about the OpenStack infrastructure extension in the short and long term, we must think about which backend storage Glance should use. The OpenStack community has developed the most commonly used storage backend either within proper OpenStack-supported storage services such as *Swift Object Storage* and *Cinder Block Storage* or a third-party extension such as *Ceph* storage based on **RADOS Block Device** (**RBD**), VMware storage, and even AWS **Simple Storage Service** (**S3**) storage.

This variety of backend storage options might increase the paradox of architectural choices, but it raises the question of which needs we would address from business and capacity perspectives. As an example, Glance configuration enables multiple storage backends in the same configuration layout that can be customized to instruct the imaging service to use an existing block storage pool to attach images directly and reduce the time of waiting to download from scratch. A massive Ceph infrastructure with dozens of object storage nodes can be leveraged if they are operational by dedicating a Ceph pool for production images, making use of existing resources granted by the stable driver integration between Glance and the Ceph backend.

> **Important note**
>
> It is important to understand your requirements before selecting any type of backend, as not all backends will support the same features. For example, within the *Antelope* release, the OpenStack Glance team included a new feature in the block storage backend – *Cinder* – by allowing the extension on the running attached volumes. The Ceph storage extension will be discussed in more detail in *Chapter 5*, *OpenStack Storage – Block, Object, and File Shares*.

Conversely, more interesting use cases would leverage Swift as a storage backend for Glance snapshots and templates. That can be considered a safe internal backup scenario. Going further, increasing your **fault tolerance** (**FT**) domain can cross public cloud services such as the AWS S3 backend.

One of the most impressive progressions in the Glance service is the wider list of supported images. The *Antelope* release came with at least 11 supported formats, including RAW, QCOW2, VDI, VHD, ISO, OVA, PLOOP, and Docker, and great additional formats dealing with Amazon public cloud compatibility are AKI, AMI, and ARI:

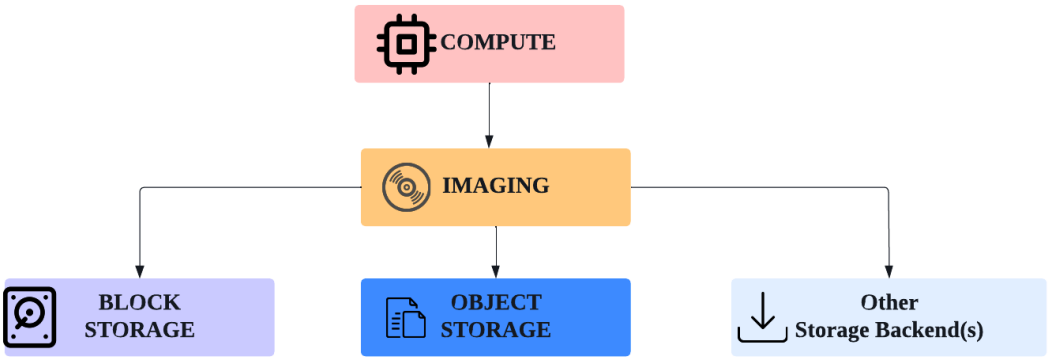


Figure 1.1 – Glance supported backends

Swift has been one of the most preferred options to store Glance images in large deployments, which will be highlighted in the next section.

Swift – the object storage service

Together with Nova, **Swift** came with the first-ever OpenStack release, *Austin*, back in 2010. Object storage has added great value to enterprises that resolve several storage challenges, unlike the traditional persistent storage design. Swift itself has been a revolutionary service at a time when cloud technologies just started to warm up. As per the nature of the object storage model, objects are stored in a flat hierarchy. Most of the main usage of Swift in a given OpenStack deployment is to perform archiving and backups. Many services around the OpenStack ecosystem that deal with storage are compatible with the Swift storage backend either for direct storage, backup purposes, or even both.

As we will unleash a few of the latest additions on the Swift service in *Chapter 5*, *OpenStack Storage – Block, Object, and File Shares*, a brief revisit of object storage common specs are listed as follows:

- Developed with no **single point of failure** (**SPOF**) by design
- Exposes the HTTP REST API for common object management
- Highly scalable and a good fit for workloads demanding boosted performance

- Designed for eventual consistency
- Highly available by design and can scale easily horizontally through inexpensive hardware

Now, let's look at the next storage option: the Cinder service.

Cinder – the block storage service

The other storage offering in the OpenStack ecosystem and counted as a core service is the block storage service named **Cinder**. Cinder has been developed to maintain independent life cycle operations of instance volumes (as virtual disks). The Cinder API provides an exhaustive list of different volume and snapshot operations, including create, delete, attach, detach, extend, clone, create images from volumes, create volumes from images, and create volumes from snapshots. With recent OpenStack releases, more operational capabilities have been developed to support backups, restoration, and volume migration.

Manila – the shared filesystems service

File-sharing solutions have been an emerging storage option for enterprises. The need to have a collaborative central storage brain for access facility and simplicity of management has grown in the latest decade in IT infrastructure either on-premises or in cloud environments. The OpenStack community did not miss the opportunity since the *Liberty* release, introducing the **Manila Distributed and Shared File Systems** solution. Coming from the same logical workflow as Cinder, Manila has not been included as a core OpenStack project due to its limited capabilities. As per the *Kilo* release until *the latest releases*, Manila had proven itself as a stable component within the OpenStack infrastructure, allowing enterprises to gain access to self-service file-sharing for different hungry resources and clients.

Once again, several storage vendors have kept the momentum of contributing to the Manila service by developing more drivers to support their storage backend hardware. Following the same steps as Cinder, we might find an exhaustive list of supported file share drivers by Manila, including CephFS, LVM, Hadoop HDFS, EMC, IBM, and many others. A full list of supported drivers can be found on the Manila OpenStack web page: `https://docs.openstack.org/manila/latest/configuration/shared-file-systems/drivers.html`. More interesting updates have been included in the latest releases for the Manila service, including security, backup share integration, and increased level of accessibility, which will be discussed more in *Chapter 5*, *OpenStack Storage – Block, Object, and File Shares*.

Neutron – the networking service

The previous citation might be arguable if we consider `nova-network` still being used! On the other hand, since the *Grizzly* release, the **Neutron** project has become a more **Networking as a Service** (**NaaS**) pillar in the OpenStack ecosystem. That makes sense due to the approach taken by

the community to provide an independent networking service and not embedded, as is the case with the Nova compute service. The transition from legacy `nova-network` to Neutron has been tougher for companies willing to migrate from the old days to the new network era led by Neutron. Although we will keep the light on Neutron in the next parts of the book, a section will be dedicated in *Chapter 6*, *OpenStack Networking – Connectivity and Managed Service Options*, to discuss the more advanced features of Neutron.

The motivation to adopt Neutron is fairly obvious considering the rise of this project to solve many `nova-network` limitations and open new networking capabilities. Compared to `nova-network`, which provides basic networking in the OpenStack ecosystem (mainly limited to interaction with the compute service), Neutron brings several aspects and features that can summarized as follows:

- Simplified self-service for projects to provision ports, subnets, networks, and router objects
- Advanced networking features can be deployed in no time, such as firewalls, **virtual private networks** (**VPNs**), and load balancers
- More support for complex network topologies
- Vast ways of third-party network solution integration
- A standalone service making architecture less complex to support **high availability** (**HA**)
- Effective service for all sizes of deployments

With the emergence of hardware and software based on **software-defined networking** (**SDN**) solutions, the OpenStack Neutron team has taken the lead to engage more support on the SDN part, resulting in the development of Neutron drivers to integrate some SDN famous implementations such as OpenContrail and VMware NSX.

Since 2013, within each OpenStack release, the Neutron project has been considered one of the services with the most number of commits and feature development. *Chapter 6*, *OpenStack Networking – Connectivity and Managed Service Options*, will iterate in depth about more Neutron capabilities and connect the pieces with SDNs within the latest OpenStack releases.

Ceilometer – the telemetry service

The telemetry service was introduced in the OpenStack ecosystem in the *Havana* release. Code-named **Ceilometer**, the telemetry service has emerged with a pertinent mission: record, gather, and monitor resource utilization metrics across the OpenStack infrastructure. Unlike older OpenStack releases, the latest Ceilometer versions include only metric collection features, leaving alarming and notification functions to another dedicated project code-named Aodh.

Aodh – the alerting service

As part of the monitoring services chain, enabling users to trigger custom notifications based on telemetry events has become a required addition to keeping an eye on deployed project resources. As mentioned in the *Ceilometer – the telemetry service* section, the alarming feature has been separated from Ceilometer, and **Aodh** is dedicated to firing alarms based on configured rules and set thresholds.

> **Important note**
>
> Ceilometer is designed to send collected metric data to other destinations, including a non-OpenStack open source project named *Gnocchi*. The Gnocchi project formulated the time-series **Database as a Service** (**DBaaS**) term. In this case, a dedicated database interface is no longer required to store and query metrics.

Horizon – the dashboard service

Having a **graphical user interface** (**GUI**) to operate your OpenStack environment would definitely make your administrative tasks much simpler. Surprisingly, the OpenStack first releases did not bring a GUI initiative to fire a few clicks from a dashboard to assist in managing the infrastructure resources. It was only in the *Essex* release that **Horizon** was born to accommodate the first core project's operations, such as the creation of object containers, operating a life cycle of instances, managing images, and a basic layout on project users' management, formerly named tenant users. The need to include more control from the dashboard has been felt with the extensive inclusion of more services, and, within the latest releases, we might notice that the Horizon experience is much more sophisticated. Built with the Django framework and supporting most of the OpenStack APIs, the majority of the core services' resources can be operated through Horizon. On the other hand, more advanced operations such as customizing networking configuration can be fired only using the OpenStack CLI. Additionally, do not expect that Horizon will automatically include an extra service beyond the core ones. It is not a limitation, but, for modular design reasons, installing a new service would require the addition of a new Horizon module associated with it and defined as a *panel*. Horizon, with a good reference of the CLI, would be sufficient to administer your cloud if running a small setup. Talking about the wider infrastructure, with dozens of projects and hundreds of resources, administering this size of setup might not be ideal using Horizon. Agile thinking would require more automation and scripting on a large-scale size. In *Chapter 2*, *Kicking Off the OpenStack Setup – The Right Way (DevSecOps)*, we will discuss in detail the automation of the OpenStack setup and how to use the same approach to manage resources within project resources themselves.

Non-OpenStack services

As part of the OpenStack control plane, we might find, as per any software architecture, other essential core components that all OpenStack services rely on: **Advanced Messaging Queuing Protocol** (**AMQP**) and database services.

Passing the message – AMQP

AMQP, the messaging queue service, is one of the other ingredients for making a modular architecture design in the OpenStack ecosystem.

For each request hitting a specific OpenStack service API, the message queue server will make sure to communicate messages between all engaged components (processes) to accomplish the request asynchronously. A variety of open source AMQP software can be used throughout your OpenStack deployment if you are biased about using one of them; it can just be simply specified during the setup to have it running and communicating with different components. RabbitMQ, Qpid, and ZeroMQ are supported message queue solutions; RabbitMQ is the most used one. Being an expert in AMQP operations is not a requirement for a successful OpenStack operation day, but a few main points should be taken into consideration when dealing with a message queue in OpenStack. We will see in later chapters how OpenStack services depend on the message queue from an architectural perspective, giving importance to such services running on the highest possible availability. The other aspect, having some adequate familiarity with how messaging concepts work, would be more than helpful for troubleshooting days. *Chapter 3*, *OpenStack Control Plane – Shared Services*, and *Chapter 7*, *Running a Highly Available Cloud – Meeting the SLA*, will highlight in sequence the internal workflow of AMQP in OpenStack and how to maximize its availability.

Storing the state – the database

The other fundamental OpenStack control plane service is the database. Similar to the AMQP service, each OpenStack service will require a database connection to store its different states. The database in the OpenStack ecosystem defines persistent storage that can be consulted and updated by all services upon each request. The database topic from an architectural perspective has been a *hot* topic mostly due to security concerns. As mentioned earlier, in the *Nova – the compute service* section, there is a tendency from that learning curve to reduce any possible insecure access to the database or services running in different hosts that could potentially present a security risk. The other side of the database sensitivity topic, similar to the AMQP service, is the availability dilemma. Besides the risk of failures, OpenStack architects and administrators will need to reiterate through the other traditional database anomaly once it starts growing: performance. *Chapter 3*, *OpenStack Control Plane – Shared Services*, and *Chapter 7*, *Running a Highly Available Cloud – Meeting the SLA*, will cover how to ensure a healthy start of the database for larger OpenStack deployments. *Chapter 9*, *Benchmarking the Infrastructure – Evaluating Resource Capacity and Optimization*, will bring the motivation behind the database benchmarking routine for a better state of performance.

Other services

The latest OpenStack releases have demonstrated goodwill to OpenStack users to deploy and operate an OpenStack private cloud infrastructure not only with very stable core services but also with additional ones. Compared to older releases, several incubated projects have been in on-off mode, making the decision to adopt some of them in your existing deployment insecure. Different insights from different project initiatives have sculpted more confident software within the latest OpenStack releases ready to support more features and services in your cloud environment. Some of the extra stable services are listed in the following table in the order of integration through different OpenStack cycle releases:

Service	Release	Description
Heat	Havana	Orchestration PaaS extension on top of the IaaS provided by OpenStack. Heat abstracts cloud resources into code – also known as **domain-specific language** (**DSL**). That comes when cloud operators leverage the power of code templating to automate things, reduce human error, and increase the agility of resource management. Heat uses **Heat Orchestration Templates** (**HOTs**), in YAML or JSON format, allowing users to define the whole infrastructure to run their workloads from code.
Trove	Icehouse	DBaaS allows users to automate the provisioning of relational and non-relational scalable databases with the minimum overhead of database tasks such as patching, maintenance, and backup through provided automation.
Sahara	Juno (incubated in Icehouse)	**Elastic Data Processing as a Service** (**EDPaaS**) provides a means of orchestrating structured and unstructured data processing and analysis software and infrastructure on top of OpenStack, such as Hadoop and Spark clusters.
Ironic	Kilo	**Bare Metal as a Service** (**BMaaS**) allows users to provision infrastructure directly on a physical machine, thus no hypervisor layer is involved.
Murano	Kilo	**Application as a Service** (**AaaS**) enables developers to speed up application deployment and publish into an application catalog that can be browsable and ready for deployment in an automated fashion.
Designate	Liberty	**DNS as a Service** (**DNSaaS**) handles **Domain Name Service** (**DNS**) entries such as DNS records and zones.
Barbican	Liberty	**Secret Manager as a Service** (**SMaaS**) is designed to centrally store different types of secrets, such as passwords, keys, and certificates, to be used by other OpenStack services in a secure way.
Zaqar	Liberty	**Messaging as a Service** (**MSaaS**) allows users to provision and manage multi-tenant messaging and notification queues.

Service	Release	Description
Magnum	Liberty	**Container as a Service** (**CaaS**) orchestrates a set of containers through a supported **container orchestration engine** (**COE**) such as Docker Swarm, Kubernetes, and Mesos.
Octavia	Liberty	**Load Balancer as a Service** (**LBaaS**) provides a load balancing function. Octavia can be counted as an enterprise-class load balancer solution. This service has become an attractive feature that can be used much more easily compared to the Neutron LBaaS driver.

Table 1.2 – Additional OpenStack services

After reviewing the core and incubated OpenStack services, we can now walk through different functional requirements that would construct what a private cloud can offer as services.

Constructing the motivation

Before building a picture of your future OpenStack environment, it is necessary to have business goals that align with the outcome of the OpenStack solution. As we have seen briefly in the previous sections with the latest updates in the OpenStack glossary, you might feel how overwhelming it could be with newly developed services and features. Obviously, the new image of OpenStack looks more robust than ever, with many hands of **Anything as a Service** (**XaaS**) opportunities. The array of choices might prevent architects and operators from starting a proper OpenStack infrastructure exercise if the scope of your business is not defined from the first day of drafting the design. As with existing major public providers such as AWS, Azure, or GCP, the OpenStack deployment should be considered as a way of investment in the long run. Expecting operational and cost savings can be felt only at later stages. Being strategic is essential to take advantage of the OpenStack solution if used properly from the early days. Hundreds of enterprises have switched gears to use OpenStack in their environment, but not all of them had a successful journey and soon dropped it. For this reason, understanding in which use case your business will fall ensures a safe start to your OpenStack journey.

Since 2010, OpenStack usage has taken different facets whereby companies have invested in the OpenStack ecosystem, followed the milestone generation, and adopted it. After 14 years, we can see a plethora of deployments where OpenStack shines with a respectful history of success. The next sections will broadly list a few common use cases of implementations.

Application development accelerator

The modern **software development life cycle** (**SDLC**) takes a more mature direction by enhancing the stages of its release and automating the whole chain, from simple code commits to different types of testing. Although **Continuous Integration/Continuous Delivery** (**CI/CD**) methods and tools emerged slightly before the rise of the cloud era, delivering a well-tested software product has faced

a new challenge: *resource limitations*. Although the wording *limitation* is not exactly the term that would describe this challenge, from a developer's perspective, it means there are no available resources to continue or perform specific unit, smoke, or integration testing. Preproduction environments are even more challenging to validate in all stages if a business will not allow the product to go live without provisioning exactly the same environment as in production for the sake of testing. Resource instrumentation and virtualization have resolved this issue, but only partially. With a lack of automation and an engine of operation, the whole chain still does not save costs and reduce the **time to market** (**TTM**). That is where OpenStack has been engaged for several hundred enterprises: accelerate product releases by running their CI/CD pipelines in a multi-tenant OpenStack environment.

Cloud application enabler

If you are planning to provide what most developers dream of (developing from scratch!), then you are in the right corner. Whether the developed application is a web pet store, online reservation assistant, or flight reservation engine, wrapping the application in its proper infrastructure can be done in a fast and clean way using Heat, for example. As the application will be defined as *code* in a HOT file, you will not need to worry about the application configuration consistency as well as its degree of scalability if autoscaling is declared in the template. As OpenStack services will take care of all the automation overhead, application owners can focus on the next level of the application business to find more spots for improvements rather than wasting effort on resource management and operation overhead.

The other trait of using OpenStack, especially within the survey from the *Kilo* release, is the trending cloud-ready application. OpenStack enables a fast application publishing process in different ways. As container technologies adoption increases, OpenStack has included container services to support a few famous container orchestration engines, such as Docker Swarm and Kubernetes, via the OpenStack Magnum service. 76% of users in the *User Survey* from the *Juno* release showed interest in the OpenStack and containerization marriage. A well-known example of a vast OpenStack environment is CERN. Based on OpenStack, CERN is running over 300,000 cores, with a big chunk of it being Kubernetes workloads (over 500 clusters) within the Magnum space.

> **Important note**
>
> More details on the conducted OpenStack survey can be found at `https://www.openstack.org/user-survey/2022-user-survey-report`.

The Murano application catalog service is another OpenStack service that facilitates application publishing in no time based on containers.

HPC supporter

It might be confusing to mix the *HPC* term in the context of OpenStack. Eventually, OpenStack does not access the hardware to configure compute power and increase hardware performance. On the other hand, it enables designing your commodity hardware in a way to grow horizontally when hitting more compute, storage, and network demand. In the HPC context, the main focus is on augmenting the underlying hardware capacity, and the cloud will secure the multi-tenancy to access resources in the most optimized way.

Surprisingly, in the early days of OpenStack when only Nova and Swift existed, the **National Aeronautics and Space Administration** (**NASA**) and Rackspace were the initiators of the first OpenStack production environment running HPC. More research organizations have since adopted the OpenStack ecosystem to deliver compute and storage at a large scale to researchers and scientists, but again, selecting the hardware layout and planning ahead is still required to feel the added value of OpenStack either via virtualized infrastructure, bare metal, or both. OpenStack enables all methods of orchestration for HPC environment deployments.

Network functions virtualization moderator

Only a few enterprises in the telecommunication industry realized the opportunities of virtualizing network services in the early days. Following the same steps as the server virtualization experience, enterprises started investing in the new trend of running their edge network services on standard machines and providing functions such as routers, proxies, load balancers, and firewalls to consumers as VMs. This became a major asset in responding quickly to network changes and feature demands. **Network functions virtualization** (**NFV**) relies heavily on compute power that can run on commodity hardware and virtual environments, whereas in the traditional way, physical appliances running each network function must exist. In the OpenStack context, NFV will need only a compute service to expose different functions. As per the *Kilo User Survey*, more than 12% of organizations use OpenStack in the networking and telecommunication industry. As Telco adopts OpenStack for such a purpose, Nova will be your assistant but with a strategic eye on compute resources and placement to make it happen.

Big data facilitator

Volkswagen AG and BMW have shared their feedback on using OpenStack for big data and analytics, and the outcome was valuable assistance of the ecosystem orchestrating all the different services and resources to accomplish some large data analytics tasks. Before the *Juno* release and the unleashing of the Sahara – EDPaaS – project, big data resources could be orchestrated using HOTs to provision a cluster in no time. Sahara was unleashed to simplify the deployment of such resources in more granular and ready-to-go ways, considering the type of the data processing framework, versions, size of the cluster, desired topology, and so on. The big data scope in OpenStack can be wider when thinking out loud by using some advanced features of the OpenStack services, such as storage and networking. Companies keep showing interest in automating their **Extract-Transform-Load** (**ETL**) pipelines internally by leveraging the OpenStack big data service. If you consider your business to be

bound to a Hadoop provider, for example, make sure that performance is a key specification for your underlying hardware that Nova will use.

Private cloud service provider

Private cloud environments could be the first main motivation when reading the definition of OpenStack for the first time in the old releases' documentation. Being private will limit the services offered to end users within the perimeter of your organization or, technically, *behind the firewall*.

The term *private cloud* might sum up all the use cases previously. The only difference is that previous OpenStack implementations can stay isolated within your perimeter or serve on-demand needs.

Public cloud service provider

Going public! Why not? Back in the early 2000s, Amazon internally launched a private elastic compute cloud to enable infrastructure agility for developers and users running the giant online e-commerce website and other services selling millions of items worldwide. **Simple Queue Service** (**SQS**) was launched and exposed to the public as the first cloud service, then later S3, followed by the EC2 service. The idea started with object storage named S3 and the EC2 service. Originally, NASA and Rackspace initiated and immensely contributed to software code to create software operating a private cloud initially for elastic computation and object storage capabilities. That is how Nova and Swift came to be. The first release attracted early-bird companies seeking innovations and started contributing to the new open source and soon cultivating more experience on market models for XaaS products. Rackspace launched its *public cloud* version offering compute and object storage services initially. This was followed by many others adopting a public cloud powered by OpenStack, such as Open Telekom Cloud, Cleura, VEXXHOST, and many more. Lately, it feels like the world runs on OpenStack with the hashtag `#RunOnOpenStack`. With its widespread implementation with big names around the world, the OpenStack community has initiated the concept of *OpenStack public cloud passports*. The idea behind it is to act as a major public cloud provider by providing OpenStack services in different regions and locations worldwide. The program ensures a massive hub of collaboration between different OpenStack public cloud providers, enabling users to roam between more than 60 **availability zones** (**AZs**) worldwide. The `openstack.org` website keeps updating the public cloud locations that can be filtered through the OpenStack public cloud marketplace page: `https://www.openstack.org/marketplace/public-clouds/`.

Picking up the pieces

Independent of which use case your future OpenStack deployment will start with, mastering the art of its ecosystem is a vital step before investigating any further design and architecture options. OpenStack covers more than 1 million lines of Python code! At this stage, there is no need to worry about learning how the code works: far more important is to take a reverse engineering approach: understanding the ecosystem interactions. Simply understanding the interrelation of core services gives you the keys to the kingdom of the art of design. Starting with the basics, the foundational components will open

more doors to the next ones if we master the core ones correctly. So far, we have revisited core services of OpenStack that exist with each release and had a brief scan of the incubated projects, particularly within the latest releases. Next, we will select a common workflow that illustrates the different steps and pieces to spawn a VM. This type of workflow will demonstrate how different OpenStack core services interact and will involve almost all the components.

> **Important note**
>
> The listing of which services are counted as core ones might differ between different resources and documentation. Telemetry, scheduling, and dashboard services could be considered optional services. The current assumption is based on real-world deployment that leverages the usage of those services from the start of the journey.

The first iteration demonstrates a high level of the building blocks of different OpenStack core services that include compute, imaging, identity, block storage, monitoring, scheduling, dashboard, and networking services:

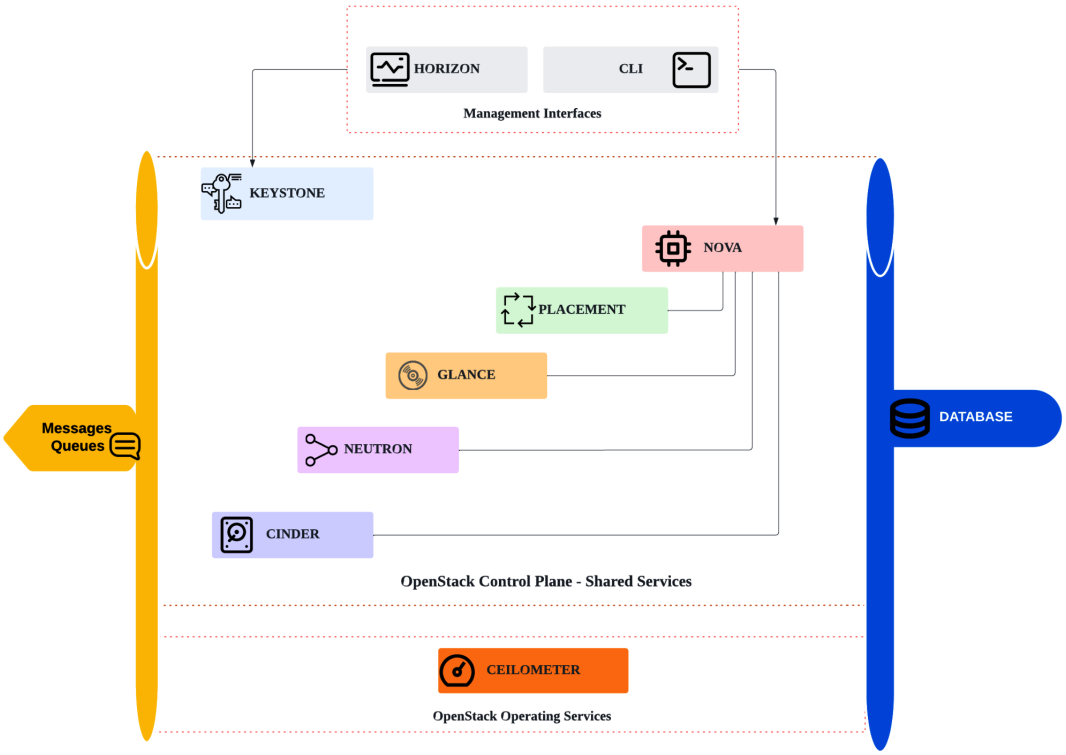


Figure 1.2 – High-level services interaction spawning instance workflow

It is important to take note of the main blocks before jumping into the granular presentation of the spawning process as illustrated in the previous diagram. By providing the Horizon dashboard or via the CLI (similarly, via APIs), an authentication process will be fired first by challenging the user request with login credentials that will be handled by the identity service, Keystone. Once validated (an authentication token will be provided for the whole session and subsequent requests), the request will reach the Nova API to take the next steps. Nova will interact with each of the following services counted as *mandatory* to successfully spawn an instance:

- The imaging service, Glance, queries an image to start the instance.
- The networking service, Neutron, acquires networking resources.
- The block storage service, Cinder, allocates storage volumes.
- The scheduling service, Placement, allocates available compute provider resources.

Note that the whole chain of interactions passes through the API of each component. For each inter-component request, the core database will be consulted for read and object updates. As per spawning an instance, a record of a new VM instance will be created in the instance and used for reference with other services in the subsequent steps. As per the instance request, Nova will interact with the queuing message service as well – for example, to request an instance launch (for the Nova compute process).

As part of this workflow, any API endpoint, scheduling, imaging, identity, and shared components such as database and queuing services are counted as part of the OpenStack control plane.

Important note

The control plane and data plane are ways of conceptual separation in complex system architectures. The terms will be defined in more detail in *Chapter 3*, *OpenStack Control Plane – Shared Services*.

Now, let's zoom in and explore the workflow in a more granular view so that we can build a whole spawning instance picture. As demonstrated in the previous part of the chapter, each OpenStack service includes subcomponents working together to serve a specific request. Obviously, the API part exists in each service. Remember that is the *gate* of each module in the OpenStack ecosystem. Each exposed API service (providing an HTTP-based RESTful API) reaches other components and, subsequently, subcomponents via the queuing message service bus, as shown in the following workflow diagram:

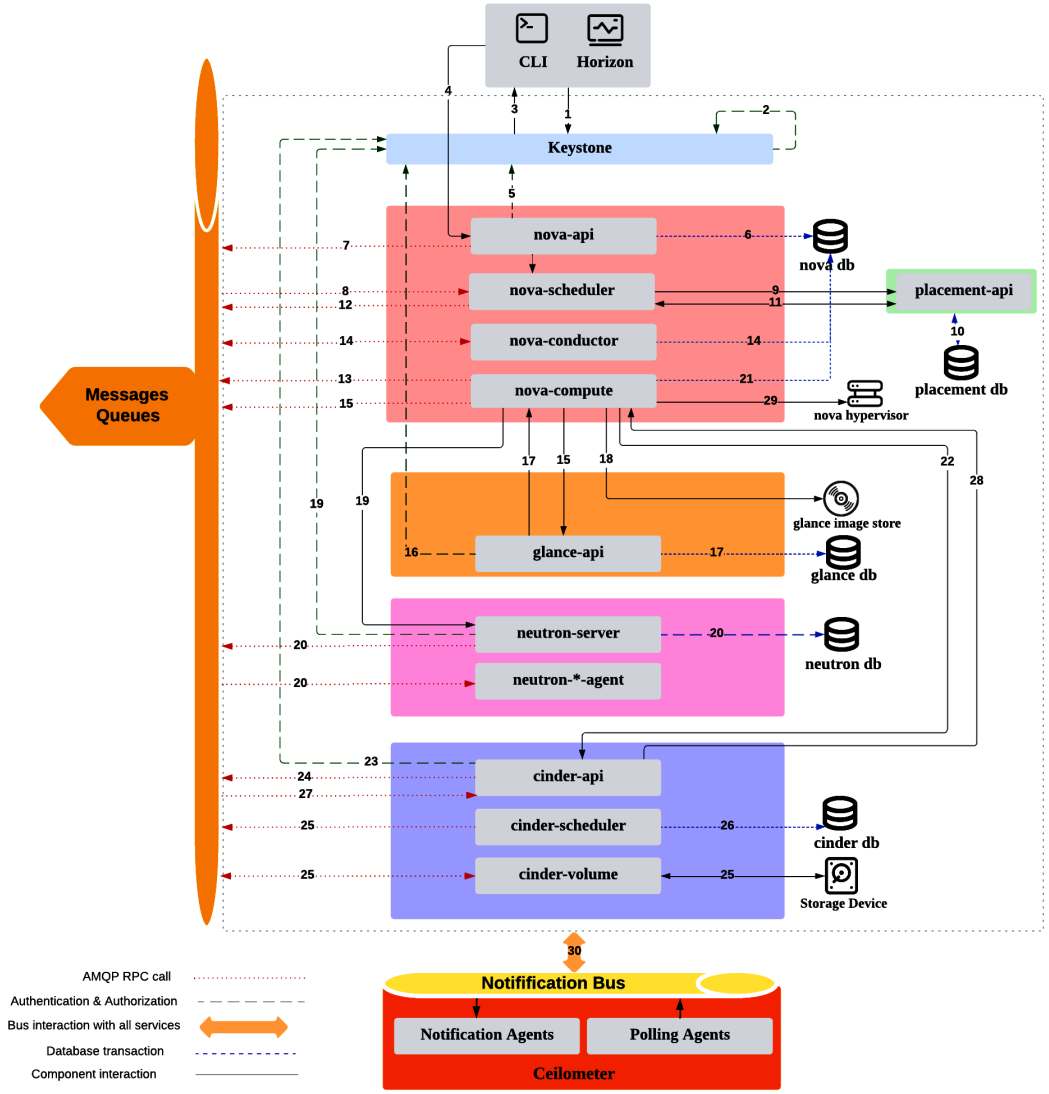


Figure 1.3 – Low-level services interaction spawning instance workflow

Starting with the identity service, Keystone should be aware of the available services in the architecture setup to authorize any further interaction. Keystone holds a service catalog containing all API endpoints of different services.

The previous diagram illustrates each step by including different components and their respective subcomponents:

1. A REST API call will be initiated by using Horizon or the CLI and reaches the identity service Keystone endpoint.
2. Keystone challenges the request, gets the user credentials (a form of username and password), and validates the authentication via its API.
3. The validated authentication generates an authentication token (an `auth` token) back to the requester user. The `auth` token will be cached for the specific user to be used for subsequent API REST calls between the rest of the services.
4. As our request is a *compute* one, the next station that will be reached is the compute service API, `nova-api`. Again, the form of the request is a REST API, forwarded by Keystone from its catalog. You can think of the catalog as a *service map* where Keystone locates each service via the registered endpoint. In this case, the REST API encapsulated with the `auth` token will be sent to the compute service endpoint.
5. The compute API, the `nova-api` subcomponent, checks the `auth` token coming with a request reaching the compute endpoint. At this stage, the `auth` token will be validated in return by the identity service, Keystone. The purpose of this secondary interaction with Keystone is to inform the compute API in the request which permissions and roles users will have and for how long the `auth` token will stay alive before generating a new one (noted as the token expiry period).
6. Once validated, the first database interaction will take place by `nova-api`, which creates a new VM object that specifies the different parameters in the request. Bear in mind that each service will have its own logical database to update the state of different resources. In the current scenario, the Nova service holds its dedicated database schema for different state updates and read and write operations.
7. Now, we come to the next interaction of another subcomponent of the Nova service, `nova-scheduler`, to specify which compute node will host the new instance. The `nova-api` subcomponent will contact the scheduler with such a request via the communication hub: the messaging queue service. This nature of the operation is processed via a **Remote Procedure Call** (**RPC**) call, noted as `rpc.call`, which will be published in the message queue and remain there till `nova-scheduler` picks it up.
8. The `nova-scheduler` subcomponent takes the lead on fetching the most convenient compute node to host the new instance. In our highlighted basic core design, the Placement service will be engaged in the process of selecting the best compute node(s).
9. The `nova-scheduler` subcomponent reaches `placement-api` to query its compute resource provider. Note that the Placement service is still considered a new service as an extension of the other services for tracking their resources, such as compute, IP pools, and storage, expressed as resource providers. Since the *Stein* release, it has been extracted from the Nova ecosystem and runs on its own. In our workflow, we can consider the usage of

Placement in the *prefiltering* stage before the final selection at the Nova scheduling level. The Placement service also exposes its own endpoint in the Keystone catalog. Hence, it will validate the `auth` token via the identity service, which updates the token header and returns it to the `placement-api` process.

10. The selection of hosts is determined via the Placement service by querying its own inventory database and, at the first iteration, returns a selection of compute nodes and any traits associated with them.
11. By applying its filtering and weighing parameters at the returned result by the `placement-api` component, the `nova-scheduler` process will come up with the right compute node and reach `placement-api` in the second iteration to claim the resources for the instance.
12. The `placement-api` component adjusts its state records in the database, and `nova-scheduler` updates the state of the VM object in its database with the compute node ID and its hostname and then submits a message in the form of `rpc.cast` to the messaging hub. This message will be picked up by `nova-compute` to launch the instance in the designated hypervisor host ID.
13. To communicate with the next subcomponent, `nova-conductor`, `nova-compute` will publish the next `rpc.call` call in the message queue to start the launch of the instance with details for preparation, such as CPU, disk, and RAM specs. Note that `nova-compute` can be running in different process occurrences in different compute nodes. Only one particular `nova-compute` occurrence will publish `rpc.call` to `nova-conductor` based on the compute node chosen in *step 8*.
14. If we recall the `nova-conductor` addition in the Nova service architecture, the main purpose of this service is to play a *gate role* between the `nova-compute` process and the database to minimize the database exposure and hence reduce its attack blast radius. As we are aiming for a large OpenStack deployment, the `nova-compute` component will be detached from the controller node (holding the main control plane services) and will be dedicated to different physical hypervisor nodes. If one of the compute nodes has been compromised, spreading that security issue to the database will be more difficult due to the logical and physical separation when processes interact with the database. The `nova-conductor` component will be the one interacting directly with the Nova database to read the instance details, such as compute node ID, its requested flavors, memory, CPU, and disk claims. The state will be reported back to the messaging queue service by an RPC publish call.
15. The `nova-compute` component picks up the published message and subsequently reaches the next service component by sending a RESTful API to the image service, Glance: the `glance-api` process. The request contains a form of `GET API` request for the image details encapsulated in the requested image ID information.
16. Glance will provide the requested image ID through its image-registered URL. But first, as we are reaching out to another service in the ecosystem, Glance will contact the identity service, Keystone, to validate the `auth` token again. Assuming the token session has not expired yet,

Keystone will validate the token, and in the same way as Nova in *step 5*, the identity service will check the user role and permissions before it updates the token header and sends it back to the Image API.

17. The Glance component is considered much simpler than Nova, as there is no interaction with a message queue in this step. The `glance-api` component simply checks the Glance database (running its own dedicated database schema), retrieves the metadata of the requested image ID, and reports back to `nova-compute` with the image URL format.
18. The `nova-compute` component reaches the image store and starts loading the provided image via the returned URL.
19. We now come to the next service to interact with: networking. The `nova-compute` component sends a RESTful API request to the Neutron service, via `neutron-server`. As discussed earlier, Neutron (by naming convention in its own architecture) does not involve an explicit subcomponent name with the `api` suffix. The `neutron-server` subcomponent is the *API gate* that will deal with all API requests. Running the same identity cycle and assuming that the token has not expired yet, `neutron-server` will forward the API request to Keystone, where it gets validated, updates the token header with different roles and permissions, and sends it back to the `neutron-server` process.
20. The `neutron-server` component checks the networking item's status and respectively creates the requested resources encapsulated in the network parameters' API call. Note that the Neutron service varies between different types of requests and hence exposes an array of workflows depending on the request complexity. We will cover other workflow scenarios dealing with other network controllers, agents, and plugins, such as **Open vSwitch** (**OVS**), in *Chapter 6*, *OpenStack Networking – Connectivity and Managed Service Options*. For example, once a port is created for an instance, `neutron-server` will publish an RPC call in the message queue reaching out to the **Dynamic Host Configuration Protocol** (**DHCP**) agent. The `neutron-dhcp-agent` process will invoke its associated DHCP driver to reload its host's file entries via the `dnsmasq` process. Once the instance is booted, `dnsmasq` is ready to pick up the instance request and send a DHCP offer for the final instance network configuration. This scenario also assumes a network and subnet are created where the instance will be attached, and that is specified in the instance creation request.
21. Once networking parameters are acquired for the new instance request, `nova-compute` reaches the database and updates the VM object network state record.
22. The last service to accomplish the instance creation is to reach out to the block storage service. Following the same way as previous services, `nova-compute` sends its RESTful API request to the Cinder API, the `cinder-api` subcomponent.
23. In turn, the Cinder API process will get to the identity service where Keystone validates the `auth` token. Assuming that it has not expired, Keystone returns the updated token header once validated with the roles and permissions set for the requester.

24. As with Nova and Neutron, Cinder has more subcomponents that would take a few steps further before the final admission of the request back to the `nova-compute` process. Cinder involves its own scheduler as well by publishing an RPC call to the messaging queue service.
25. The `cinder-scheduler` subcomponent picks up the request and locates available resources for a new volume candidate list based on the requested volume specifications (volume size and type). Note that Cinder also provides another process, `cinder-volume`, that interacts with `cinder-scheduler` whenever a new read or write request is initiated within a specific storage provider. In this case, `cinder-volume` will be interfacing with backend driver methods to generate a candidate list that will be posted in the message queue and picked up later by the `cinder-scheduler` subcomponent.
26. The `cinder-scheduler` subcomponent updates the state of the volume ID with its associated returned metadata in the Cinder database.
27. The `cinder-api` component picks up the published message from `cinder-volume` and responds to `nova-compute` via a RESTful call.
28. The `nova-compute` component receives the created volume metadata in the API request. By default, if no other specific hypervisor virtualization driver is being configured, Nova will execute the `libvirt` daemon and proceed by creating the instance on the designated compute node.
29. For the final stage of the instance spawning workflow, the created volume should be mapped by the `cinder-volume` process. That is when the virtualization driver, in this case, `libvirt`, will make the volume available to the instance by firing a `mount` operation. The way it is performed depends on the storage provider and protocol, but a common case is passing the volume path to the hypervisor, which will mount it to the instance as a virtual block device.
30. The last piece of our initial workflow is the ingestion of the instance metrics in the Ceilometer service. It is vital to watch what has been spawned by checking the status of the created instance. Ceilometer provides two ways of collecting data metrics for instances: *polling agents* or *notification agents*. Collected metrics will pass through a Ceilometer pipeline transformation for further manipulation and prepare this data to be published. The latest releases come with an array of choices to store the published data in the Ceilometer database, which can be accessed through the Ceilometer API, Gnocchi, or even a simple file store.

Zooming in on the communication between the different core components is essential before drafting a first design layout of your private cloud, which will be detailed in the next section.

Architecting the cloud

Identifying the initial use case of your OpenStack environment is a major key to a successful start of the cloud journey. With a vast array of features, especially within the latest releases, the cloud management platform has become more mature than ever, but that might be overwhelming. Sidetracking the purpose of your OpenStack implementation would increase management complexity and potentially put your business at risk if such investment is not carefully accompanied by a vision. For this purpose, drafting

your design into different iterations will help you avoid being blocked by *the paradox of choices* and unleash a clear short-term vision toward the longer one. As the first design iteration, we will simply identify our conceptual OpenStack model, followed by the logical architecture. Once drafted, we will walk through the practical implementation and put some numbers together to reflect our first deployment picture in the hardware.

Drafting the conceptual design

Based on the latest launching instance workflow, we have settled on the core services that should exist in our first OpenStack implementation. Naturally, any services that will be extending our basic setup later, such as PaaS services, will rely on the services underway that construct mainly our IaaS layer. During the next design phases, we will consider a generic OpenStack use case that would be suitable for internal usage as a private cloud to support organizations with elastic IaaS resources, boosting cloud application development, enabling data analysis, and helping with public hosting workloads. Beyond the core services, other mature ones will be included in this design iteration, as summarized in the following table:

Cloud service	**Service role**
Compute – Nova	Manages VM life cycle UI-enabled
Imaging – Glance	Manages VM image files UI-enabled
Object storage – Swift	Manages objects for persistent storage UI-enabled
Block storage – Cinder	Manages VM volumes for persistent storage UI-enabled
Networking – Neutron	Manages network L2 and L3 resources UI-enabled
Data metrics – Ceilometer	Collects resource data metrics for monitoring UI-enabled
File share – Manila	Manages scale-out file share systems UI-enabled
Identity – Keystone	Authentication and authorization
Dashboard – Horizon	GUI
Scheduling – Placement	Prefiltering for resource providers and traits

Table 1.3 – Initial OpenStack services

Summing up those services into one conceptual diagram will captivate our next logical design iteration and narrow down the distribution of the services across different cluster roles of the OpenStack implementation:

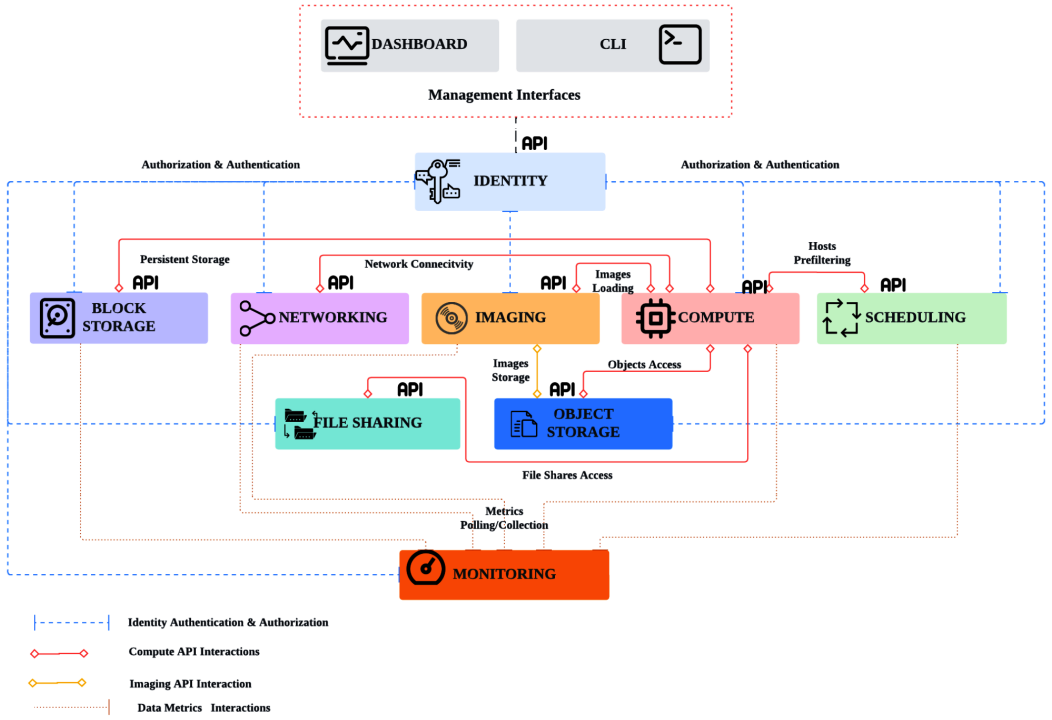


Figure 1.4 – OpenStack services interaction workflow

The previous diagram illustrates a high-level presentation of our first OpenStack architecture draft that counts several services to host a variety of workloads for end users. We can proceed with our design by developing a logical layout in the next section.

Drafting the logical design

The logical design iteration is a vital step that should be carried out with more attention. More brainstorming points will be raised at each level and within each service of the design to formulate an initial draft that can be implemented with more confidence. The other key consideration is to run our services for both the control and data planes in a minimum setup of HA. That can be empowered by clustering the controller and compute nodes in the subsequent iterations. While working on the cloud roadmap, it is essential to reflect on the growth aspect. As your business needs are identified, the first

logical design should be mapped with the first explored requirements. Trying OpenStack even with the minimum installation can be helpful to familiarize yourself with its basic ecosystem components. On the other hand, a major consideration of the first deployment for future successful experiences is to think of *growth*. As mentioned earlier, OpenStack is developed with modular software architecture; reflecting the same approach in the logical design followed by the physical one will absolutely make your cloud shine. Consider the following guidelines to begin with:

- Define roles for the OpenStack system in both the control and data planes
- Isolate roles starting with the minimum number of nodes and sufficient hardware
- Think ahead of the possibility of component services failing
- Use software that the cloud operations team is most familiar with, such as queuing message and database common services

Throughout the first iteration of our logical design, an incremental approach will be taken, starting with the following roles:

- A pair of controller nodes running the control plane, including the database and queuing message.
- One compute node running the hypervisor layer based on **Kernel-based Virtual Machine** (**KVM**).
- MySQL with the MariaDB engine and RabbitMQ will be chosen as the database and messaging queue services, respectively. Most of the OpenStack-documented implementations use MySQL and RabbitMQ as almost a religious matter.
- HAProxy and Pacemaker will be running in each controller node for HA, load balancing, and clustering. Database HA will be implemented through Galera multi-master replication. RabbitMQ instances will run on their native clustering mode based on queue mirroring across both cloud controller hosts.

Important note

The first logical design should not be overwhelming as it is not a final draft ready for production yet. *Chapter 7*, *Running a Highly Available Cloud – Meeting the SLA*, has been dedicated to going through each layer of the future implemented control and data planes in detail.

The first logical design proposal can be drafted by using the previously listed software tools and the cloud controller and compute roles, as follows:

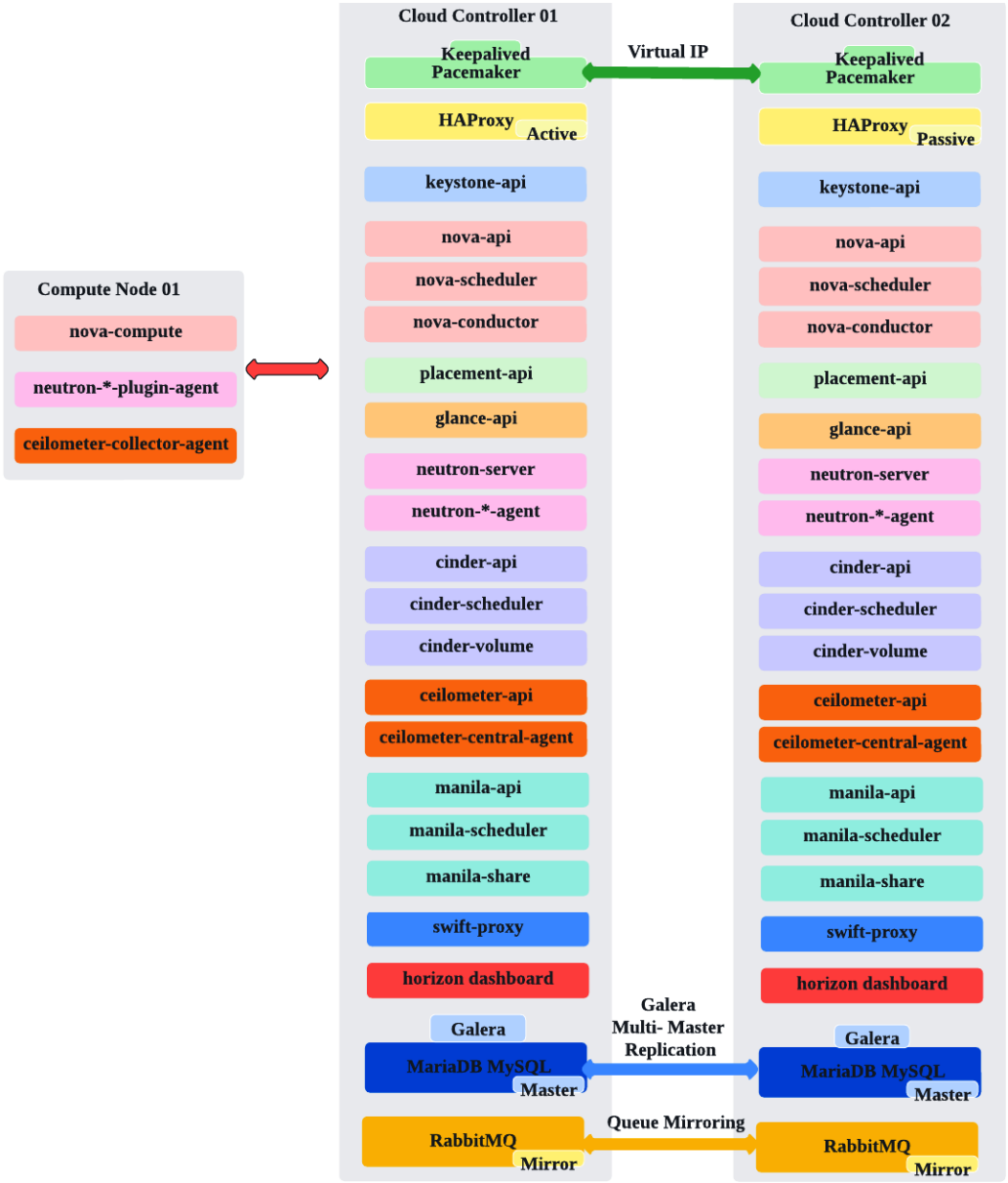


Figure 1.5 – High-level logical design of OpenStack components for deployment

Once the logical setup of different components and roles has been drafted, we will need to identify how the OpenStack nodes should connect.

Connecting the dots

Another essential aspect during the logical design phase is to provide a consistent networking layout between the different roles of each OpenStack entity. There are gazillions of ways to connect different pieces in an OpenStack ecosystem; the following layout demonstrates a network segmentation approach:

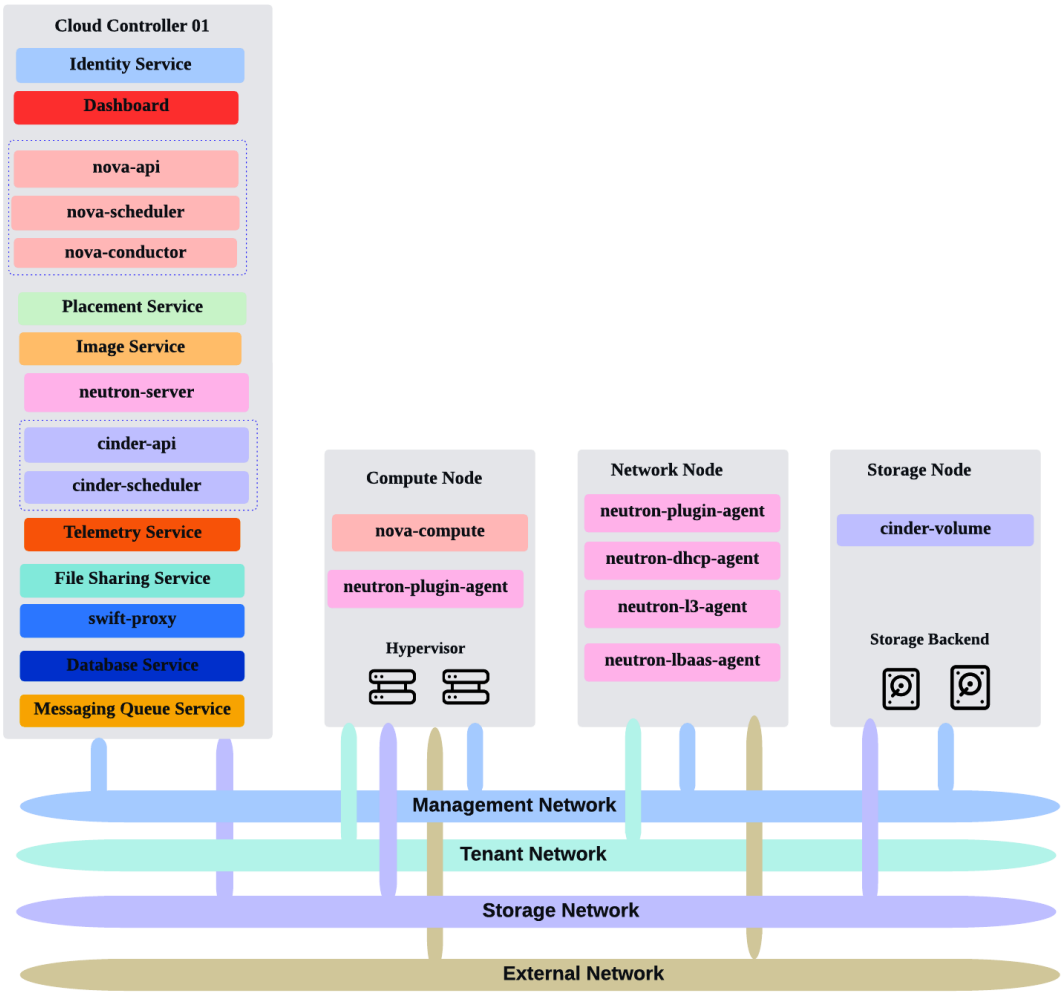


Figure 1.6 – Network segmentation OpenStack nodes

The isolation of different types of networks will increase security access through networking segmentation. The trade-off is the extra complexity, but security comes first. The other opinion on this choice is also the performance question, as dedicating a separate segment for each type of traffic would save bandwidth significantly.

The previous diagram illustrates four types of networks denoted with the following conventions:

- **External network**: Connecting to the outside world. OpenStack APIs can be reachable from the public as well as private backbone within a global organization network. The external network will provide routable IP addresses. This type of network will be part of the data plane as it will expose or direct traffic instances within the underlying infrastructure. It is essential to keep security configuration tied at this level by fronting the network with load balancer devices or appliances.
- **Management network**: As part of the control plane, this provides interconnectivity between all the various nodes in the OpenStack environment. Services such as the database and queuing message will be plugged into this network to be reached by the compute nodes. The management network does not expose anything to the outside world and should be seen as a private or internal network (you might find the naming convention of *API network* in other sources).
- **Tenant network**: As users will require their virtual networks, the tenant network (referred to as a *guest* or *overlay network* in other sources) will be dedicated to handling instance traffic. There are a variety of options for this type of network that could be attached to SDN capabilities in the network node. *Chapter 6*, *OpenStack Networking – Connectivity and Managed Service Options*, will highlight the scope of virtual overlay networking in more detail. The tenant network can be considered as part of the data plane.
- **Storage network**: As part of the data plane, the storage network will connect compute, and storage cluster nodes.

> **Important note**
>
> Instances can use a direct physical network instead of virtual ones empowered by Neutron through SDN, which would assign floating IPs to access the outside world. This model is referred to as a *provider network* that would connect network and compute. More details will be highlighted in *Chapter 6*, *OpenStack Networking – Connectivity and Managed Service Options*.

Enumerating the different types of networks earlier will help the selection and reservation of different ports and network cards of each host per role in the long-term setup. It is still possible to combine more than one network in the same physical connection. However, the continuous expansion of the underlying infrastructure will hit the limit of the physical capacity of the combined segments, leaving a variety of issues with bottlenecks and network performance anomalies. Planning and preparing in advance will save an immense amount of time and effort in modifying the networking layout once network performance starts boiling. If a network node is dedicated in the next iteration, for example, the controller node will not need to connect to the tenant and external networks.

Drafting the physical design

The next section will assign values to our parameters defined in the previous logical design. However, understanding a few key concepts and practices beforehand will save plenty of effort and costs. Let's start with a capacity-planning demonstration.

Preparing for capacity planning

Capacity planning comes at every stage of a new infrastructure project, either within in-house hosting or for the development of a simple application. The cosmos of capacity-planning practices is based on forecasting and predicting how many resources an IT infrastructure would require to respond to business needs. In the OpenStack context, once your business case has been defined, the process of capacity analysis is narrowed to a specific set of resources that should exist. If you are planning to host a mixture of generic web hosting applications and data analysis workloads, there are certain considerations to be planned in terms of hardware sizing and the technology that will run the workloads. NFV would require more attention when selecting the hardware as it can be a performance eater.

Building the capacity planning for the OpenStack case can be summarized as follows:

- **Operate with elasticity**: Be able to respond and pull more ubiquitous compute resources when needed in the case of a failure or load increase using automation.
- **Expect to fail**: Underlying resources should be ready to be replaced immediately without the need to spend time on fixing and reconfiguration efforts during incidents.
- **Track for growth**: Available capacity over the course of production can be variable. As we are in an on-demand model, the expected growth is not forcibly linear. Regularly track the usage of your underlying infrastructure to plot your cloud usage and update the capacity roadmap.

Another key foundation that is highly appreciated by large infrastructure management is the adoption of **Information Technology Infrastructure Library** (**ITIL**) practices. By reflecting on the IT infrastructure that will run the OpenStack cloud environment, ITIL methodologies will definitely refine a strategic process to identify a complete cycle of your capacity management under the ITIL service design umbrella. If your organization has already rolled out ITIL practices, feel free to reuse and apply them in the cloud journey. From a **Cloud Service Provider** (**CSP**) perspective, having a tactical approach to managing the underlying IT infrastructure to align with business needs and user demand and take full control of the financial aspect is a *must-have*.

> **Important note**
>
> ITIL is a framework presented with a set of practices and methodologies to standardize, manage, and optimize IT services offered in a given business. ITIL has evolved into four different versions. All the versions emphasize common core concepts around the business service, such as the service design pillar targeting capacity management.

Mapping the land

As a first iteration, it would be ideal to dig into a few parameters that would generate some numbers to start the journey. In order to estimate our hardware and configurations, we can think of a reverse approach. Instead of looking at how many instances the first deployment could accommodate, we should see this from the end user's eyes: *Which instance flavors could be offered to run which specific workloads?* An instance flavor is a set of specifications defined as a characteristics template in the compute service, including the number of vCPU cores, amount of RAM (including swap), and root disk (including ephemeral size).

> **Important note**
>
> More advanced flavor specs can be customized and created in OpenStack to deploy instances that require specific workloads by scheduling the Nova service to use a set of compute nodes, such as support of certain CPU architecture or intensive workloads for HPC.

A simple approach is to spread the population of compute resources into an average flavor model that could run generic workloads. As a start, an instance flavor could be suitable for testing tenant environments with a couple of vCPUs and 1024 MB of RAM capacity. Defining the baseline flavor model will enable us to determine the next ones by doubling the size of capacity in the next iteration. Keep in mind that there are gazillions of ways to define the set of flavors, but, most importantly, identifying the starting one will help to classify the next sizing in the series.

Depending on your captured business needs, hence the type(s) of workload(s), you will have a clearer picture of how much hardware capacity will be required in each compute node you plan to invest in. With every combination of flavors, putting the density of resources in the compute box will help to measure whether any wasted room has been left behind so that it can be used within a new flavor. For example, if a compute node could support 40 medium and 10 small-sized instances and still have some room left, create a new flavor with the gapped size of 1 vCPU and 512 MB of RAM to be added to the compute node flavor catalog.

The following use case will consider a business initiative to run instances with versatile types of generic workloads that include instance flavors:

Flavor	vCPU	RAM (MB)	Disk (GB)
Tiny	1	512	10
Small	1	1024	20
Medium	2	2048	40
Large	4	4096	80

Table 1.4 – Instance flavors list

The first capacity-planning roadmap will assume 200 VMs as a starting point with the first set of compute nodes. The following sections will conduct an estimation for different hardware specs, including CPU, RAM, storage, and network for each compute node.

CPU estimation

A proper calculation of the compute power will depend significantly on the technology and model supported by the hardware. For this reason, we will define a list of hardware assumptions for our use case as follows:

- GHz per physical core = 2.6 GHz
- Physical core hyperthreading support = use factor 2
- GHz per VM (AVG compute units) = 2 GHz
- GHz per VM (MAX compute units) = 16 GHz
- Intel Xeon E5-2648L v2 core CPU = 10
- CPU sockets per server = 2
- CPU overcommit ratio = 16:1
- Oversubscription: Disabled
- Hyperthreading: Disabled
- Operating system overhead: 20%

> **Important note**
>
> OpenStack compute allows the overcommitment of resources for CPU and RAM. The technique of overcommitment aims to maximize the usage of resources by leveraging sharing between VMs running in the same hypervisor machine. For example, running 16 vCPUs per 1 physical CPU in a hypervisor host is denoted as a CPU overcommitment of 16:1.

Let's start with the number of virtual cores:

(number of VMs x number of GHz per VM) / number of GHz per core

```
(200 * 2) / 2.6 = 153.846
```

A rounded decimal estimation will result in 154 vCPU cores.

Now, let's take into account the operating system overhead running in the compute node of 20%, as follows:

```
154 + ((154 * 20)/100) = 184.4
```

A rounded decimal estimation will be 185 vCPUs.

Adding the overcommitment ratio 16:1 parameter to estimate the actual physical CPU on the compute node can be calculated as follows:

```
185/16 = 11.5625
```

A rounded decimal estimation will result in 12 CPU cores.

As a starting point, a compute node of 12 CPU cores would host 200 instances with a *small* model flavor.

Memory estimation

The following assumptions would be required to align with the previous CPU estimation:

- 1024 MB of RAM per instance for the small flavor
- 8 GB of RAM maximum dynamic allocations per VM
- Compute nodes supporting slots of 2, 4, 8, and 16 GB sticks
- RAM overcommit ratio of 1:1
- Operating system overhead of 20%

For 200 VMs with small-sized flavor instances, a RAM estimation is given as follows:

```
200 * 1024 MB = 200 GB
```

Adding the 20% operating system overhead gives us the following:

```
200 + ((200 * 20)/100) = 240 GB
```

A 1:1 overcommitment ratio will determine the actual RAM size needed per compute node:

```
240/1 = 240 GB
```

That would require an amount of 240 GB of RAM in our compute node to accommodate 200 small flavor instances.

> **Important note**
>
> The default Nova compute configuration comes with an overcommitment value of 1:1.5. Be aware that, unlike CPU overcommitment behavior, memory overcommitment can affect the instance performance if not enough room for the swap memory is planned in advance. As a good practice, with 1:1.5, configure the swap memory to be at least double what is provided.

Storage estimation

As per storage, physical compute nodes should acquire enough storage capacity with a multitude of options by providing the instance's root disk from the compute nodes' physical disks themselves or via attached storage devices.

Considering 200 VMs and a small size flavor instance of 20 GB root disk, a compute node should acquire an amount of *200 * 40 = 800 GB*.

Depending on the operating system requirement for disk space and other factors with caching and swapping configuration, an estimation of between 900 GB and 1 TB of storage will grant a secured disk allocation for all instances (think of swap and extra caching disk operation overheads).

Another aspect that can be considered for more flavor customization is the storage type specs. That was not mentioned in the previous flavor catalog table, but, as part of the capacity management exercise, when the business changes the scope of the nature of the workload, extra classes of flavors can be added to expand that list, such as high-performance storage and **input/output operations per second** (**IOPS**) specs. Storage devices have various specifications to deal with certain use cases and patterns; for highly intensive workloads that require fast reads and writes, **Solid-State Drive** (**SSD**) disks should be considered in the hardware list, followed by a new custom flavor presenting the SSD specifications.

Networking estimation

Networking capacity planning comes in two categories:

- Network topology, and switching and routing layers to provide sufficient IP addressing of the underlying cloud layer
- Instances and overlay networking functions running on top of the infrastructure

The first pillar would require a consistent network layout that would reflect which node would acquire a private IP, a public one, or both. Obviously, when putting the nodes together, as per network design, the first step is to assign each host its private IP from a given private IP pool. Components that would require access to the outside world, as defined in the logical draft, would use public IPs. That can be granted via frontend devices or appliances such as load balancers or routers via **Source Network Address Translation** (**SNAT**) and **Destination Network Address Translation** (**DNAT**) mechanisms.

The second pillar captures a few factors of network capacity estimation targeting *undercloud* resources, which includes mainly the following assumptions:

- Minimum bandwidth of 50 Mbits/sec per virtual interface per instance
- Association of one floating IP per instance
- Association of one floating IP per NFV overlay function – virtual routers
- 10% reserved floating IP for future use

Another essential aspect that should be considered when planning for networking is the network interfaces that will be configured in each compute node. As we covered a high-level design in our logical architecture draft, a few networks will be created and assigned to each interface in each host. This architecture will involve switching L2 configuration, typically via **Virtual Local Area Network** (**VLAN**) aggregation. Once basic configurations are in place, running some benchmarking tools is necessary to gather a few metrics on the bandwidth capacity in each switch port and the compute host interface. That will help to get at-a-glance network performance information for the hop between the compute host and the main physical switch port. As the physical bandwidth will be shared between different virtual network instance interfaces, the first value of the switch port and compute node physical interface should be acceptable.

The next consideration is taking into account three network interfaces cabled to each compute node. One of the network interfaces attached to the tenant network (associated with the VLAN in a dedicated switch port) will be cabled using a 10 GB physical link that will serve 200 VMs, giving 50 MBits/sec for each.

> **Important note**
>
> L2-specific vendors' network performance varies from one device to another. Make sure to measure network traffic and supported **Maximum Transmission Unit** (**MTU**) sizes when designing your network segmentation and VLAN configurations.

Physical network failure and performance should be highlighted as early as possible instead of waiting to be alerted. Including whole configurations and resetting cabling and interfaces when running in production does not sound like the most brilliant approach. One of the most popular networking techniques that would save cost and performance ahead is **network bonding**. Bonding supports two different modes: *active-backup* and *active-active*. The first will keep running only one network interface as active while the others are in the backup state, whereas the second involves the *link aggregation* concept – **Link Aggregation Control Protocol** (**LACP**) – where traffic will be load balanced through different interfaces.

To avoid network performance bottleneck surprises, the same approach can be taken with physical switches. A common well-known technology, **Multi-Chassis Link Aggregation** (**MLAG**), would transform several physical switches into one logical one, allowing FT and exposing the best-effort bandwidth to the connected node's ports.

Empowering the network node's performance without taking care of what capabilities the physical switching could provide would not guarantee a promising experience. Gathering the physical network capacity metrics in all the various layers is vital to avoid hitting unexpected performance bottlenecks.

The second pillar of our initial network capacity planning is the floating IP pool. As Neutron will be our network master in the OpenStack ecosystem, we are expecting the network node to interact with different network resources, such as instances, virtual routers, and load balancers, without mentioning advanced SDN configurations that would overwhelm our initial estimations. Floating IPs are publicly routable (public IPs are typically what you get from an **Internet Service Provider** (**ISP**)). A floating IP request either from an instance or a network function resource such as a router or load balancer should not fail. Thus, we will estimate the pool as follows:

- 200 instances per compute node: 200 floating IPs
- 20 tenants
- 10 routers per tenant: 200 floating IPs
- 10 load balancers per tenant: 200 floating IPs

Adding 10% for future use will generate a pool of 660 floating IPs.

Mixing the land

The previous capacity-planning exercise considers a mono-compute density prototype targeting a *small* flavor. The capacity-planning study could involve more than one prototype of distributing the compute density, depending on your strategic business needs. The target layout can be extended to involve more flavors, supported in either heterogeneous or homogeneous compute density form. The `nova-scheduler` component can be configured, as we highlighted in a previous section, with advanced scheduling that works hand in hand with the Placement service to find the most optimal compute node. The same resource estimation for vCPU, RAM, and root disk can be followed to iterate through the next flavor to determine the next set of compute node groups. We will cover the filtering mechanisms to respond to compute requests in the most adequate way in *Chapter 4*, *OpenStack Compute – Compute Capacity and Flavors*.

As illustrated in the following diagram, one way to set future boxes in the most predictable fashion is to assign each available compute node a homogenous flavor:

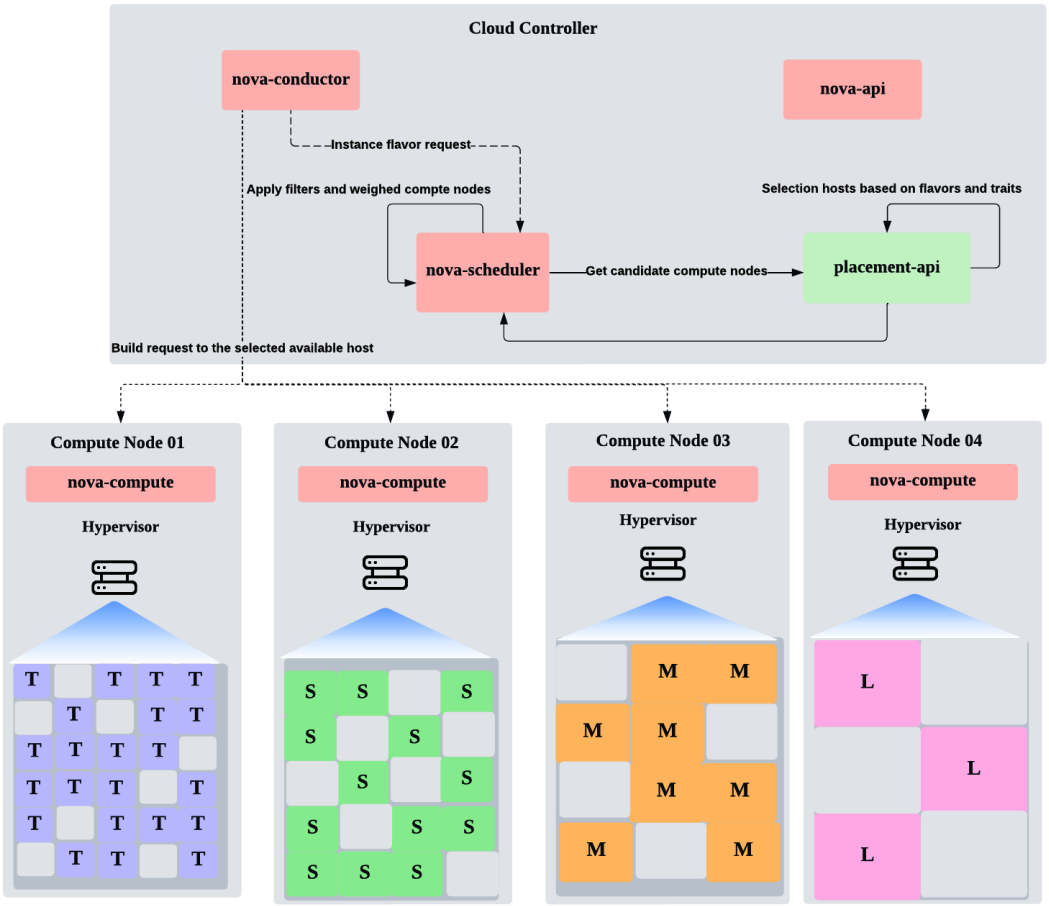


Figure 1.7 – Compute node placements per flavor and workload trait

As per the scheduling configuration together with the defined traits in the Placement service, a Nova request will reach the right available compute node to accommodate a specific workload. This model layout would classify compute nodes per types of workloads defined in your business requirements, so the compute nodes' capacity would be prepared in advance to expect the initial forecasted demand.

Summary

Being strategic is a fundamental key to making the right decisions when it comes to how to architect a complex ecosystem such as OpenStack. This chapter should lower the entry barrier to start an effective plan that meets your organization's needs. The approach taken should help you identify different stages to start your cloud journey. There is no exact rule of thumb on how to design an operational OpenStack environment but templating the design patterns for each core service and sticking to the

initially collected requirements will definitely enhance the journey. This chapter went through the newest updates on core services in the OpenStack ecosystem within *Antelope* and later releases, and a few more projects were considered to be offered once the private cloud is up and running. From an architecture perspective, this chapter should be revisited during the next stages to align and update your design draft in each step. As we used an iterative and incremental approach for our future OpenStack cloud environment, the next chapter will take you to the next deployment stage from what we have in draft, spiced up with best practices for the setup process.

2

Kicking Off the OpenStack Setup – The Right Way (DevSecOps)

"In the middle of difficulty lies opportunity."

– Albert Einstein

The OpenStack ecosystem, as discussed in the previous chapter, presents a vast array of design patterns, coupled with choices of hardware and tools to deploy a robust private cloud setup. With the Antelope release, there have been no significant changes to the core architecture system, but a few parts have been updated, with some additions and features. But that comes with the cost of increased complexity when it comes to the deployment and operation stages. The good news is that the OpenStack community, since its earliest releases, has created different approaches to avoid an arduous OpenStack operation experience. The key mechanism for this task is *automation*. There is, without a doubt, a high risk of getting into endless troubleshooting if the automation craft is overlooked. There are several tools maintained by vendors such as Puppet, Ansible, Chef, and a few others that are used in thousands of production deployments. To get an OpenStack setup up and running is not the goal of the journey but, rather, the deployment journey itself. That is where we will take advantage of automation and agile tools to bring the OpenStack operation to the next level, by adopting a DevOps path. Although that sounds like an amazing step toward a robust deployment process, it is still not complete without incubating security aspects from the early days of the deployment. Developing and operating the infrastructure process should be reimagined, from a DevOps picture to a DevSecOps one.

This chapter presents the backbone of the future deployment and operational processes for an OpenStack private cloud setup by covering the following topics:

- Revisiting the DevOps philosophy behind the OpenStack deployment life cycle
- Rethinking DevOps and introducing DevSecOps to see how it fits into our journey
- Discovering a modern approach to deploying OpenStack in containers
- Iterating through automation tools and preparing for deployment
- Running a first iteration of an OpenStack deployment via Continuous Integration/Continuous Delivery (CI/CD) tools
- Preparing for a large-scale deployment via code and Continuous Integration/Continuous Delivery (CI/CD) pipelines
- Integrating security checks in each deployment step in an automated fashion

Removing silos – DevOps

The term **DevOps** has been popping up for almost a decade, pushing organizations to rethink how to structure their internal teams to achieve what DevOps promises. Dozens of definitions of the word *DevOps* can be found everywhere, but it can simply be summarized as a *silo remover*. DevOps comes with the philosophy of bringing the main software product's direct contributors to the same table, including operations, testers, and developers. The aim of this movement is based on a few principles that motivate organizations to implement them practically – knowledge and responsibility sharing, respect, automating everything, monitoring continuously, embracing failure, and designing for reusability. However, DevOps has never been a tool or software but, rather, a set of principles that define processes and workflows to boost a company's productivity. What makes this movement shine takes us back to the simple definition – *silo remover*. As operations and developers are not hindered by the wall sitting between them, more collaboration and knowledge are shared toward the same goal – decreasing the time to market with the highest possible quality and degree of efficiency. Without the existence of many tools, reflecting this idea in real life would be more difficult. **Continuous Integration** (**CI**) and **Continuous Delivery** or **Deployment** (**CD**) tools are examples that have contributed immensely to reflecting that idea. Another set of tools that gives more trust to machines than to humans is the concept of *automation*. CI/CD tools leave all the manual and routine build, testing, and deployment stages to a few pipelines running on a server. Meanwhile, teams focus on more interesting tasks with higher business values. The rise of the cloud era has brought another giant set of opportunities to accelerate the software development cycle with automation – **infrastructure automation**. This is one of the areas that OpenStack focuses on with some of its services, such as **Heat**. Treating **Infrastructure as Code** (**IaC**) inspires the gathered teams, Dev and Ops, to bring knowledge from the traditional software development life cycle and apply it at the infrastructure level.

It is no longer a case of *that is not my business* but, rather, *it is my business too*. What really matters is focusing on the delivery of business products in a reasonably reduced amount of time, getting to market faster, and not missing out on a race with the competition. This can be summarized with a DevOps toolchain for different software deployments across the Dev and Ops stages, as shown in the following diagram:

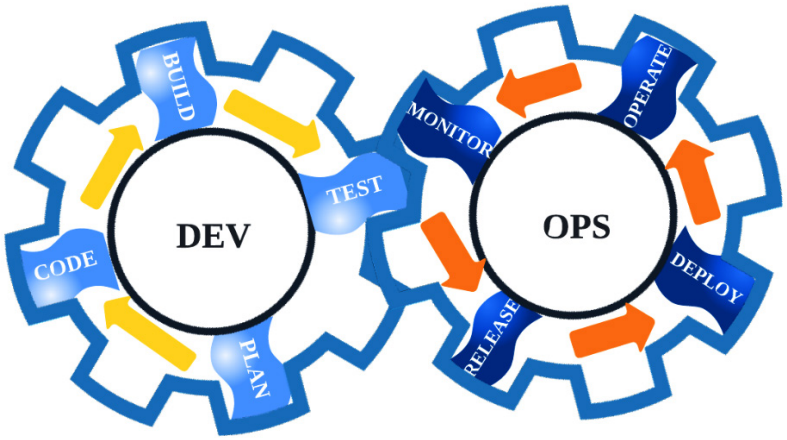


Figure 2.1 – The DevOps toolchain

By adopting the same philosophy, we will deploy our OpenStack private cloud. But before reaching that stage, we need to acknowledge an important fact – there is no rule of thumb when it comes to applying what DevOps dictates. The structure of teams in organizations may differ, and IT disciplines can be in several dedicated pods per function or merged into one. The pursuit of *DevOps happiness* can be confusing if an organization does not assign and refactor its team's structure, seek out subject matter experts, have cross-functional team members based on specialized knowledge, and so on. Some of these issues have revealed obstacles, due to the gaps left in terms of security and compliance. This is what we will uncover in the following section before bringing the pieces of OpenStack together.

Shifting to the left – DevSecOps

The *billion-dollar* idea of the DevOps philosophy is to bring people together – all disciplines, including Ops, Devs, and **Quality Assurance** (**QA**). Organizations should make it clear where silos are raised among all product contributors. That should include the security discipline in their agile ways of working. The *2021 State of DevSecOps* study (`https://www.securitycompass.com/reports/2021-state-of-devsecops/`) elaborated by *Security Compass*, a cybersecurity consultancy services provider, shows that 75% of responders (IT professionals and DevOps practitioners) agreed that security slows down the processes of product releases, leading to an increase in the time to market. That goes against what DevOps was created for. Focusing on speed to market might skip the security aspect. This *sacrifice* has triggered some scary numbers in some large enterprises, where there was an increase in the volume of data breaches or the number of cybersecurity attacks. Treating security as something

to act upon reactively instead of proactively can cost in terms of budget and time for each security incident. There are a few chief reasons that capture the root cause of why cybersecurity and DevOps are left in two different silos – more manual security intervention during the development process and a lack of knowledge of tools. Investing in security is required to comply with standards and gain trust. Sacrificing that for speed to market will certainly lead to major losses later on. That leads us to the next bridge that needs to be built between DevOps and security, forming a DevSecOps alliance. In the same way that Dev and Ops merged, security should no longer be considered an afterthought. DevOps principles can be inherited to make a security fusion possible. That is how DevSecOps has become a new trend. Imagine the whole pressure of security eased across each discipline and pointing to the same product goal, armed with automation and faster feedback.

We can sum up the right approach for our future OpenStack deployments – DevOps with security at different stages of the process, as shown in the following toolchain.

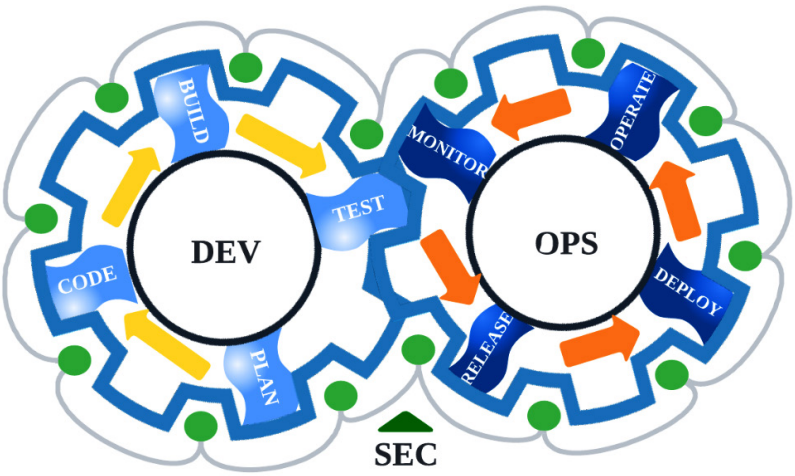


Figure 2.2 – Integrating security into the DevOps toolchain

Embracing the deployment of a complex environment such as an OpenStack private cloud would require a DevSecOps approach, and a few best practices are covered in the next section.

Securing the cloud – moving with DevSecOps

Adopting DevOps practices is definitely a big deal to stay competitive in the market, increase productivity, and bring new possibilities for innovation. Conversely, with a lack of security, any product will be at risk of losing both trust and failure. Hence, the DevOps chain must be enforced with security and compliance insurance. One of the fundamental aspects of DevOps is automation and reusability. In an infrastructure context, we are not dealing with *pets* but with *cattle*. The *pets and cattle* analogy is a famous one, with on-prem servers referred to as *pets*. If there was any change, update, or failure of that pet server, a problem would arise. They are the long-lived and indispensable pieces of the infrastructure that should be taken more care of. On the other side of the analogy, *cattle* refer to servers in large

on-demand environments such as the cloud. A cattle failure should not stop the business from running, and cattle can be replaced *immediately*. If we adopt the *cattle* analogy servers in the cloud should be instrumented to keep them immutable. That is where IaC comes into play. With one single source of truth and a shared pipeline, the *Sec* part becomes much easier to merge into the *DevOps* flow. With this capability, moving security from the right to the left becomes more feasible.

DevSecOps and OpenStack

The challenge of deploying a complex OpenStack ecosystem is made easier through DevOps, but what should not be overlooked as a cloud provider is protecting the first layer that runs a user's workload(s). As we will explore in the next part of this section, there are several ways to deploy a complete OpenStack environment from code. Emphasizing security before starting to think about the code structure is vital to not miss a DevSecOps opportunity. As a rule of thumb, any code designed for an application infrastructure is subject to vulnerabilities. Although the different modules of OpenStack services are developed with security in mind, the gazillions of ways of configuring and parameterizing within a specific network architecture should be tested for security anomalies at every stage. Before welcoming any workload on top of the OpenStack infrastructure, the cloud itself must be ready with security controls and compliant with major standards.

Coding the cloud

One of the biggest challenges of dealing with an OpenStack environment is the ways of handling its deployment and daily operations. If you consider a manual deployment for even a small environment, then in all likelihood, your OpenStack management processes, including updates, will face difficulties that cannot be avoided. That is where DevSecOps practices come in, with the fundamental mantra of IaC. A consistent approach is to treat the OpenStack ecosystem as code. The good news is that OpenStack code is provided with each stable release and is ready for deployment once configured and customized. But before we grasp a technical deployment, we should take into account a few common considerations to start building our toolchain:

- Map each OpenStack service as code per module.
- Define target systems to run OpenStack services with one specific role or several roles. For example, a compute role is defined with the Nova service deployed.
- Draft the initial configuration for each service (role).
- Think of roles as reusable code that can be deployed in different target environments.
- Make sure you integrate functional test stages between role setups.
- Consider the integration of security rules and tools from the start. Map each nominated tool in each stage of the pipeline, from coding to the release phases.
- Avoid *Big Bang* coding practices and implement smaller changes in an automated continuous deployment fashion.

- Prepare to continuously monitor the following deployment metrics:
  - **Deployment frequency**: How often IaC is deployed (daily/weekly/monthly)
  - **Change lead time**: The window of time starting from the moment code has been committed to the time it is successfully deployed
  - **Change failure rate**: The total percentage of committed IaC changes leading to system impairment or outage (including hotfixes and rollbacks)
  - **Mean Time to Repair** (**MTTR**): The time needed to restore service outage from the moment it's impaired
- Use a trunk-based development approach to facilitate code monitor changes and increase security visibility integration.

> **Important note**
>
> Trunk-based software development is a coding practice that allows developers to regularly merge changes into a common trunk code base. The trunk-based practice dictates more frequent code releases with smaller changes.

The following section will cover the different tools that will be used to deploy an OpenStack environment in the DevSecOps style.

Deploying in the cloud

Since the first release of OpenStack, several deployment tools and wrappers have been developed to assist OpenStack operators with more enhanced, easier ways to set up a fully running OpenStack environment. With the rise of system management tools such as Chef, Puppet, SaltStack, and Ansible, the OpenStack community has dedicated different channels to each system management tool to develop different classes and modules through the OpenStack ecosystem evolution. Which one of these tools we choose is down to familiarity or the technical requirements of cloud operators. In this section, Ansible will be the system management tool we use.

Ansible in a nutshell

Like any other system management tool, Ansible uses its own glossary and terms to define infrastructure components, modules, relationships, and parameters. Conversely, unlike other tools, Ansible comes with a simple architecture setup that makes it a popular choice, and it can be summarized as follows:

- It has the flexibility to handle interdependent complex service modules.
- It uses an agentless transport mechanism to connect and update target systems without the need to install additional packages. It exposes only the *master* server where the Ansible software is running.

- Modules executed on target systems are auto-cleaned once installed.
- It is rich in core automation modules to extend more features.
- The infrastructure collection code written in YAML is organized in Ansible playbooks. YAML is easier to learn and master than other languages, such as Ruby for Chef.
- Due to the simplicity of the Ansible architecture, scalability is much simpler to implement compared to other tools that require running agents.
- As a declarative method of programming, it focuses on describing the output of the desired state, rather than diving into how to do it and which steps must be taken to reach the resulting output.

When dealing with the complexity of the OpenStack ecosystem, with all its sets of components, subcomponents, and parameter flavors, it is essential to briefly skim the surface of the Ansible terminology :

- **Playbooks**: These consist of the main configuration file(s) that describes a series of actions to run on one or a group of hosts. The tasks are written in YAML and executed in order, from top to bottom, to accomplish full deployment or a configuration change.
- **Roles**: These present the organizational structure of playbooks by collecting different Ansible assets, such as tasks, variables, and modules, to deploy a service on one or a group of hosts.
- **Modules**: These are an abstract representation of a specific unit of code functionality. Modules can be written and customized to control system resources, files, and services. Ansible is shipped with some modules referred to as module libraries (core modules) that can be executed via customized playbooks.
- **Variables**: These are dynamic values used in roles and playbooks to reflect the desired configuration. Similar to programming language variables, Ansible variables enable the propagation of different states across different environments through the same role or playbook.
- **Inventory**: This is a listing of managed hosts in the target environment. Ansible uses a configuration file in the **INI** format, which defines the name and IP of the managed target host. Hosts can be declared in inventory files using different patterns by nesting hosts by role, or by specifying a series of hosts via a combination and numeric patterns.

The CI/CD system triggers Ansible to install and run playbooks across target OpenStack nodes defined in its inventory, as depicted in the following figure:

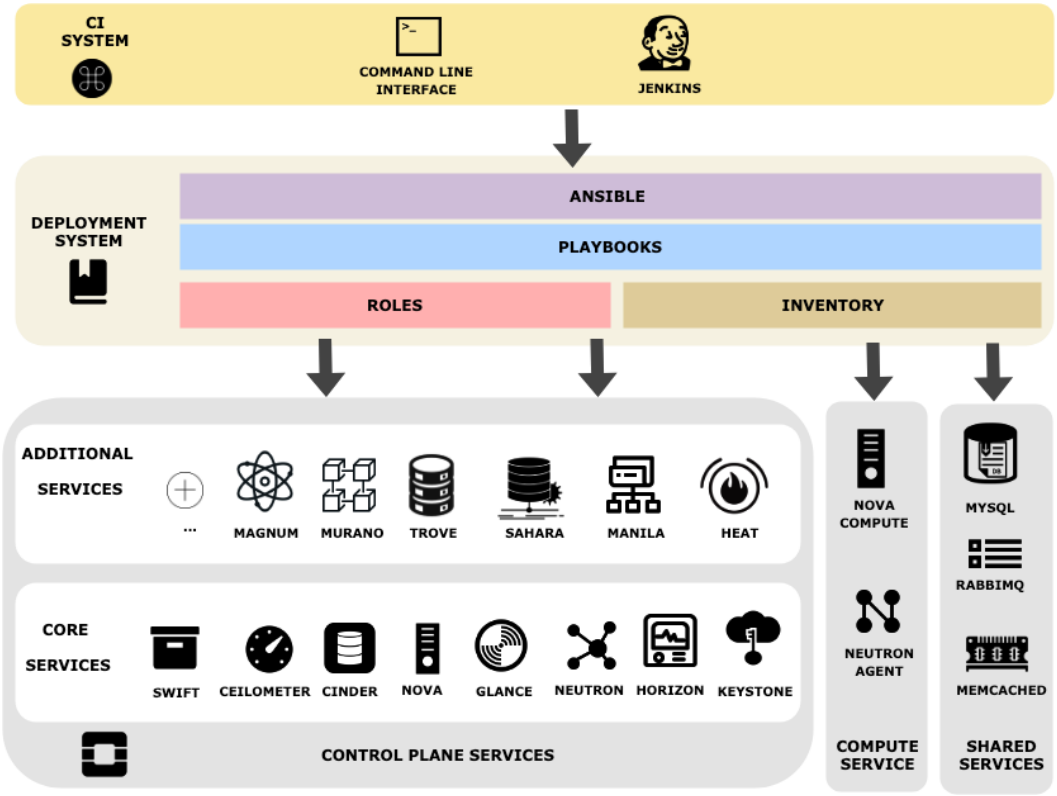


Figure 2.3 – An OpenStack environment management through Ansible

Integrating system management tools such as Ansible in OpenStack ecosystem life cycle management has introduced a significant change in the way such a complex ecosystem is managed and operated. Not surprisingly, the hunger to move quickly to adopt a private cloud setup and maximize the agility of its deployment has led to other challenges. That brings us to the root causes of OpenStack-tied dependencies between services. Thinking long term, the development of new OpenStack releases and feature integrations in an existing environment presents one of the ultimate challenges for a private cloud operation mission. Driving major upgrades has always been a blocker to seamlessly jumping to a new OpenStack release in a running production setup. That makes the upgrade case a very cautious operation and puts your **Service-Level Agreement** (**SLA**) at risk if one or several dependencies crash, due to a code compatibility version that has been overlooked. Although tests can help to identify such anomalies, a massive number of tests for each component should be in place in advance, and that could be costly in terms of resources and human interactions. The other facet of these challenges is the lack of rollback mechanisms. Rolling back a change that causes an issue will definitely bring one or several parts of a whole system down, as you wait for your management tools to redeploy, restart the affected services, and test and wait for a complete synchronization between other dependent services to run them again. The most common OpenStack deployment options involve bare-metal machines or

virtual machines, making such operational tasks heavier to roll, requiring full management of machine images, and it becomes costly when testing in an isolated environment. For example, upgrading or updating an OpenStack component version would require all inter-dependent services to be deployed in a different testing environment before propagating the change in the production one. Within the latest OpenStack releases, several organizations have been experimenting with the rise of containers for a fully containerized OpenStack cloud. Let's unleash the container dilemma in the next section and explore what opportunities are in store for OpenStack deployment and management.

Containerizing the cloud

Container technology has been around for more than a decade, and companies have started deploying their workloads in containers to take advantage of resource optimization, environment isolation, and portability. Running different pieces of your software in lightweight, self-contained, and independent containers brings more flexibility to managing a complex software system. Running in an isolated mode, operations such as upgrades and rolling back become much easier and can be performed with more confidence.

By referring to our infrastructure code, we can see the benefit of the OpenStack software architecture's modular design, which offers a great deal by taking advantage of container technology, where each module can live in a separate, stateless environment. Looking at the current state of container trends, an extended list of container and orchestration engines can be found, such as LXC, Docker, and Kubernetes. The OpenStack community did not miss the chance to adopt containerization technology early on, when containers started to become the standard for modern software development. Within the Antelope and later releases, we can find several deployment methods based on containers, combined with configuration management tools, as summarized in the following table:

Deployment project	**Container**	**Project**	**Source**
OpenStack-Ansible	LXC	Ansible	`https://docs.openstack.org/openstack-ansible/latest/`
Kolla-Ansible	Docker	Ansible	`https://docs.openstack.org/kolla-ansible/latest/`
OpenStack-Helm	Docker	Kubernetes and Helm	`https://docs.openstack.org/openstack-helm/latest/`
Triple-O (currently no longer supported)	Docker	Ansible	`https://docs.openstack.org/tripleo-ansible/latest/`

Table 2.1 – A list of OpenStack deployment tools running containers

The **OpenStack-Ansible** (**OSA**) project is one of the most widespread deployments based on LXC containers. The deployment of LXC containers running OpenStack services is orchestrated by Ansible.

In our next deployment, we will adopt another emerging OpenStack project, named **Kolla-Ansible**. Similarly to OSA, the Kolla project uses Docker as a containerization tool by building a Docker container for each OpenStack service. Besides the design differences between LXC and Docker, Kolla extends the parameterization layout, making its containers more configurable, with an array of choices between the base operating system and template engines. This is not to mention Docker's design advantages as a container technology, with the layered nature of its images, versioning, and portability for sharing capabilities.

Important note

Kolla has been integrated officially within the OpenStack subproject since the Liberty release. As per the official OpenStack definition of the Kolla project, *"Kolla's mission is to provide production-ready containers and deployment tools for operating OpenStack clouds."*

Those extra advantages make Docker more suitable for our deployment pipeline, where we will deal with a container service as an artifact that can be easily deployed through different environments, before promoting it to a production environment. As depicted in the following high-level schema, for each merged change in the code repository, the CI/CD tool builds an artifact composed of a software package encapsulated in a Docker image. The generated image will be committed to a private Docker registry, where Ansible will download it and orchestrate its setup in the designated OpenStack node.

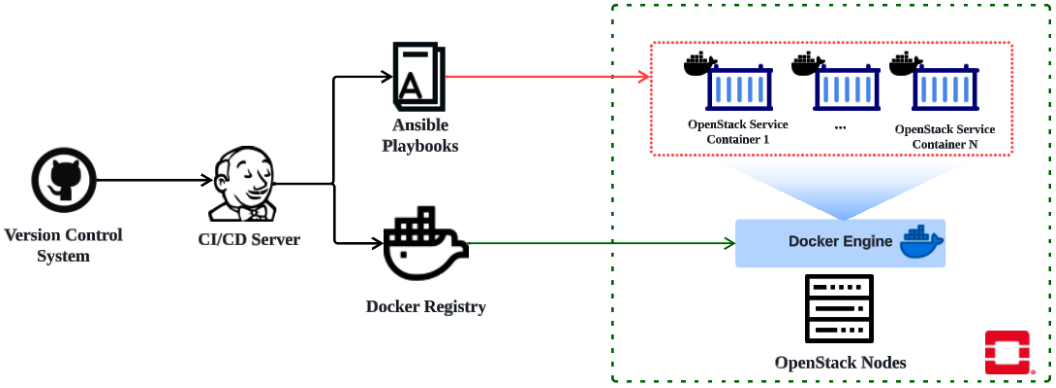


Figure 2.4 – High-level CI/CD pipeline OpenStack deployment

The other component of the deployment tool stack is the **Jinja2** templating tool. This is mainly used by Docker for the dynamic assignment of variables, based on defined parameters generated by Ansible. Jinja2 is designed mainly for Dockerfiles to support full system call interface compatibility for both RPM and DEB container distributions.

Important note

Jinja2 templating in the Kolla context provides an array of ways of building Docker images of different source distributions, including CentOS, Debian, Fedora, Ubuntu, and RHEL container operating systems that can be parameterized per OpenStack service code.

As our main motivation is to solve the complexity of dependencies and provide a simpler and more scalable development experience, Kolla is the way to go, and indeed since its official first stable release, several production deployments have been performed the Kolla way.

Building the picture

Treating OpenStack deployment as IaC will help us to inherit and use most of the software tools and processes to deliver code artifacts, ready for deployment, with more confidence. Having a robust integration and deployment pipeline is vital to ensure that our software-defined data center does not fail on production day. Modern software development techniques involve several tools that increase automation and agility and, hence, faster feedback between each deployment.

The following tools will be employed for our OpenStack infrastructure code development:

- **A version control system**: GitHub will be our code repository for the OpenStack infrastructure code.
- **A CI/CD tool**: Jenkins will be installed on the deployer machine and grant extra plugins to build and run deployment pipelines.
- **A system management tool**: The Ansible packages will be installed on the deployer machine and provide the OpenStack playbooks to be deployed inside the containers, in tandem with Kolla.
- **An image builder**: Dedicated to OpenStack containers, Kolla builds container images running different OpenStack services.

The Kolla-Ansible project code repository will be our starting point for the first OpenStack deployment, with minimum customization to have a rolling deployment pipeline initially. The official latest master stable branch of the project code can be found here: `https://github.com/openstack/kolla-ansible`.

Kolla-Ansible supports both all-in-one as well as multi-node OpenStack setups. As we're aiming for a production setup, as discussed in the initial design draft in *Chapter 1*, *Revisiting OpenStack –Design Considerations*, we will make sure our initial draft is the first deployment iteration ready for deployment.

Setting up a development environment

Setting up a development environment mimicking a production one is highly recommended to ensure faster feedback during the development stages, as well as to fix any anomalies in code before promoting it to a production setup. Although running services in a container increases the flexibility of rolling back if issues are detected after deployment into production, a best practice is to act proactively by running service containers in an isolated environment through different deployment and testing stages. Upon successful deployment in the test environment, services' containers will be promoted and tagged as production-ready for deployment.

Depending on how many physical machines will be invested in your development environments, the DevSecOps team will require an isolated environment for each member to run tests, before committing their changes to a main development branch that targets deployment in the development environment.

Preparing the local environment

Vagrant is a revolutionary tool created by HashiCorp that can create, modify, and manage visualized environments through simple Vagrant command lines. The other exciting part of Vagrant is managing a local development environment through it. Vagrant configuration is highly customizable and supports a variety of local hypervisors to run virtual machines, such as VirtualBox, HyperV, and VMware.

Our next setup will consider a local VirtualBox environment, depending on your development operating system machine; most Linux, Mac, and Windows distributions are supported and can be found here: `https://www.virtualbox.org/wiki/Downloads`.

Once VirtualBox is installed, download the base image where the OpenStack environment will run. A good practice for VirtualBox development in Vagrant is to use the same operating system distribution for both the development and production environments. The chosen operating system in the next setup is Ubuntu `22.04 LTS`. In the following wizard, we will go through the different steps to prepare the Vagrant file before starting to deploy a local OpenStack environment:

1. Generate a Vagrant file by firing the following command line:

```
# vagrant init
```

2. Modify the `box` stanza configuration section by providing the operating system image:

```
Vagrant.configure("2") do |config|
  config.vm.box = "generic/ubuntu2204"
end
```

> **Important note**
>
> Vagrant boxes can be created locally by backing up an image of any operating system, using tools such as **Packer**. Vagrant boxes can be shared and tagged with the version in the Vagrant cloud repository, as detailed here: `https://developer.hashicorp.com/vagrant/vagrant-cloud/boxes/create`.

3. The local development machine should have a minimum of hardware specs, such as RAM and CPU, to accommodate a minimal setup of an OpenStack environment, as depicted in the following Vagrant configuration section:

```
config.vm.provider :virtualbox do |vb|
   vb.memory = 8096
   vb.cpus  = 4
end
```

4. Set the virtual disk space to a minimum of 50 GB in the new `box` stanza line:

```
Vagrant.configure("2") do |config|
 ….
   config.disksize.size = "55GB"
```

Important note

Make sure to have the `vagrant-disksize` plugin installed by running the following command line – `vagrant plugin install vagrant-disksize`.

5. Optionally, add a synced folder to sync the OpenStack infrastructure code on the development machine:

```
Vagrant.configure("2") do |config|
  ….
    config.vm.synced_folder(
       "openstack_deploy/", "/openstack_deploy/"
    )
```

Important note

Make sure the synced folder in the local machine exists by creating the `openstack_deploy` folder.

6. Networking can be configured to forward port access to specific services from the development machine to the guest box. Accessing OpenStack Horizon, for example, would require port forwarding on port `80` from the guest machine, which can be adjusted as follows:

```
Vagrant.configure("2") do |config|
...
   config.vm.network :forwarded_port, guest: 80, host: 80
```

7. Start the Vagrant box by firing the following command line:

```
# vagrant up --provider virtualbox
```

8. Once started without errors, the Vagrant box is ready for access. The default login uses a Vagrant user that can be elevated to root to proceed with the next setup:

```
# vagrant ssh
# sudo -i
```

Once the Vagrant box is up and running, you should be able to start installing the required packages to deploy an all-in-one OpenStack environment.

Running the local cloud

The Kolla-Ansible project code includes different ways of installing OpenStack in all-in-one or multi-nodes. As part of the infrastructure code process, we will adjust a few settings to run an all-in-one mode on the development machine. But first, we need to install the Git package on our local development host:

```
$ apt-get install git
```

Clone the OpenStack-Kolla repository to the designated local folder setup created in the previous section. The Vagrant configuration file will make any changes to the local folder, `openstack_deploy/`, visible to the guest box. Ideally, cloning the repositories should be done from your local Git server of choice. In this example, a private Git repository is used (`ci.os`). The following command clones the local repository running on the CI/CD instance:

```
$ git clone https://git@ci.os/git/openstack_deploy/openstack_deploy
```

> **Important note**
>
> Here is a simple guide to optionally set up a local Git server running NGINX on the latest version of Ubuntu: `https://www.howtoforge.com/tutorial/ubuntu-git-server-installation/`. The local Git server uses the latest stable master branch, cloned locally from the Kolla-Ansible repository: `https://github.com/openstack/kolla-ansible.git`.

Optionally, create your development branch from the master to track local changes before pushing them to the next development branch:

```
$ git checkout -b local_dev_branch
```

> **Important note**
>
> Ideally, pushed changes should be made in a dedicated development branch of your choice before propagating them to the master branch.

Log in to your Vagrant box and proceed to install environment dependencies and create a virtual environment. Make sure to change the path of your environment on the guest host:

```
# apt-get update
# apt-get install python3-dev libffi-dev gcc libssl-dev
# apt install python3-pip
# apt install python3-virtualenv
# python3 -m venv local
# source local/bin/activate
(local)# pip install -U pip
```

Install Ansible. Make sure a minimum version of 2.16 is installed:

```
# pip install  'ansible-core>=2.16,<2.17.99'
```

Install the `kolla-ansible` runtime:

```
#  pip install git+https://opendev.org/openstack/kolla-ansible@master
```

Since OpenStack Yoga, a few additional dependencies need to be installed, which can be solved by running the following command line:

```
# kolla-ansible install-deps
```

The Kolla-Ansible repository comes with two essential files that we will edit to run our local deployment – `globals.yml` and `passwords.yml`. Copy the files to a directory of your choice, which can be created, and make sure that the running guest user owns the files:

```
# mkdir -p /etc/kolla
# chown $USER:$USER /etc/kolla
# cp -r /local/share/kolla-ansible/etc_examples/kolla/* /etc/kolla
```

The `globals.yml` file includes several settings and adjustable configurations for different OpenStack control plane services, as well as shared ones. As part of our minimal test environment, we will edit the following settings to the copied `globals.yml` file under the `/etc/kolla/` folder:

- Use Ubuntu to run as the base Linux distribution for the containers:

  ```
  kolla_base_distro: "ubuntu"
  ```

- Ensure that HAProxy is out of the scope of our initial setup in the test environment:

  ```
  enable_haproxy : "no"
  ```

- Select the network interface name to be assigned for all types of OpenStack networks:

  ```
  network_interface: "eth0"
  ```

- Dedicate a second interface for the Neutron external network:

  ```
  neutron_external_interface: "eth1"
  ```

- Assign a Keepalived floating IP, which should be reserved as a virtual IP for HA:

  ```
  kolla_internal_vip_address: 172.28.128.47
  ```

- The `passwords.yml` file centralizes all the OpenStack purposes and other services' passwords. To generate random password sets in one command instead of running them manually for each, Kolla comes with a simple command tool that generates passwords, which will be used during the deployment, as follows:

  ```
  # kolla-genpwd  -p /etc/kolla/passwords.yml
  ```

  Make sure that the generated passwords are saved in the `passwords.yml` file under your custom directory.

The other crucial file is under the `inventory` folder, which describes and instructs Kolla-Ansible on which hosts will be the target to apply the configuration. Inventory files are defined with the Ansible host group format and can be customized, based on the available hosts and assigned roles. The default available files are designed to run inventories for all-in-one and multi-hosts. To run a clean host target setup, make a copy of the files under the `inventory` directory:

```
# cp -r  /local/share/kolla-ansible/ansible/inventory/*  /etc/kolla/
```

The all-in-one Ansible inventory file will not require any changes for our development setup purposes. The format of the file is straightforward, where each host role section describes the name of the host or IP address. For the local test environment, all host roles will be configured with `localhost`, as shown in the following snippet:

```
[control]
localhost       ansible_connection=local

[network]
localhost       ansible_connection=local

[compute]
localhost       ansible_connection=local

[storage]
localhost       ansible_connection=local
….
```

Fire the `kolla-ansible` command line to install the bootstrap servers. The command will run the Ansible playbook against each host-assigned role in the inventory file. So far, all roles are gathered in the same localhost:

```
# kolla-ansible -i /etc/kolla/all-in-one bootstrap-servers
```

The following is the output:

```
TASK [Gather package facts] ****************************************************************************************
skipping: [localhost]

TASK [Group hosts to determine when using --limit] *****************************************************************
ok: [localhost]
[WARNING]: Could not match supplied host pattern, ignoring: all_using_limit_True

PLAY [Gather facts for all hosts (if using --limit)] ***************************************************************
skipping: no hosts matched

PLAY [Apply role baremetal] ****************************************************************************************

TASK [openstack.kolla.etc_hosts : Include etc-hosts.yml] ***********************************************************
included: /root/.ansible/collections/ansible_collections/openstack/kolla/roles/etc_hosts/tasks/etc-hosts.yml for localhost

TASK [openstack.kolla.etc_hosts : Ensure localhost in /etc/hosts] **************************************************
ok: [localhost]

TASK [openstack.kolla.etc_hosts : Ensure hostname does not point to 127.0.1.1 in /etc/hosts] ***********************
ok: [localhost]

TASK [openstack.kolla.etc_hosts : Generate /etc/hosts for all of the nodes] ****************************************
changed: [localhost]

TASK [openstack.kolla.etc_hosts : Check whether /etc/cloud/cloud.cfg exists] ***************************************
ok: [localhost]
```

Figure 2.5 – The kolla-ansible bootstrap servers' output

Optionally, run a few deployment checks to closely monitor the bootstrapping process and detect possible failures:

```
# kolla-ansible -i /etc/kolla/all-in-one prechecks
```

Here's the output:

```
PLAY [Apply role tacker] *******************************************************************************************
skipping: no hosts matched
[WARNING]: Could not match supplied host pattern, ignoring: enable_octavia_True

PLAY [Apply role octavia] ******************************************************************************************
skipping: no hosts matched
[WARNING]: Could not match supplied host pattern, ignoring: enable_zun_True

PLAY [Apply role zun] **********************************************************************************************
skipping: no hosts matched
[WARNING]: Could not match supplied host pattern, ignoring: enable_vitrage_True

PLAY [Apply role vitrage] ******************************************************************************************
skipping: no hosts matched
[WARNING]: Could not match supplied host pattern, ignoring: enable_blazar_True

PLAY [Apply role blazar] *******************************************************************************************
skipping: no hosts matched
[WARNING]: Could not match supplied host pattern, ignoring: enable_masakari_True

PLAY [Apply role masakari] *****************************************************************************************
skipping: no hosts matched
[WARNING]: Could not match supplied host pattern, ignoring: enable_venus_True

PLAY [Apply role venus] ********************************************************************************************
skipping: no hosts matched
[WARNING]: Could not match supplied host pattern, ignoring: enable_skyline_True

PLAY [Apply role skyline] ******************************************************************************************
skipping: no hosts matched

PLAY RECAP *********************************************************************************************************
localhost                  : ok=260  changed=142  unreachable=0    failed=0    skipped=127  rescued=0    ignored=0
```

Figure 2.6 – The kolla-ansible prechecks' output

Once the bootstrapping process is finished without any detected failures, perform the deployment of the OpenStack environment, which will take a long time to create all the containers' services and chain them together:

```
# kolla-ansible -i  /etc/kolla/all-in-one deploy
```

Here's the output:

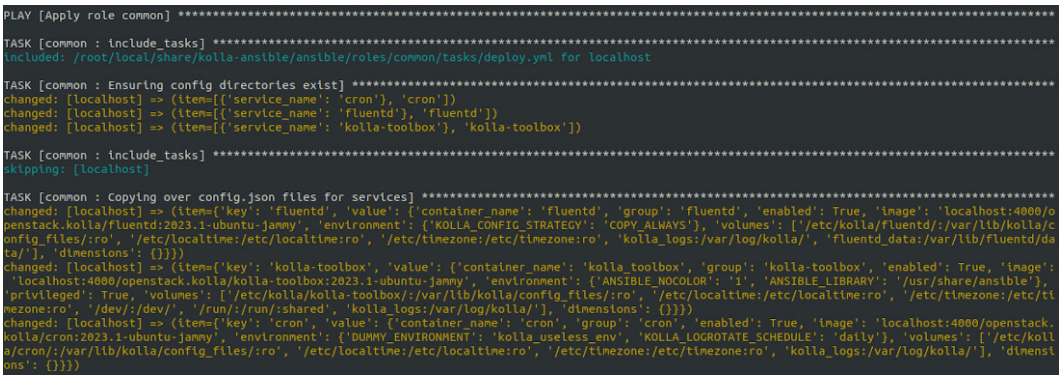


Figure 2.7 – The kolla-ansible deployment output

> **Important note**
>
> The Ansible deployment will apply the playbooks and OpenStack roles using container images from the default OpenStack Docker repository. For the next development stage, make sure to build and push further images to a local repository. The next sections and chapters will discuss how to build local container images for different services and manage them locally.

The deployment of the minimal setup of the OpenStack containers should result in zero failures by the end of the `kolla-ansible deploy` command line. The following Docker command line lists all the running containers deployed in the previous step that should be noted as healthy. Some containers will still be in the process of communicating with other system services, which will take a few minutes, before starting to evaluate the OpenStack services:

```
# docker ps
```

Here's the output:

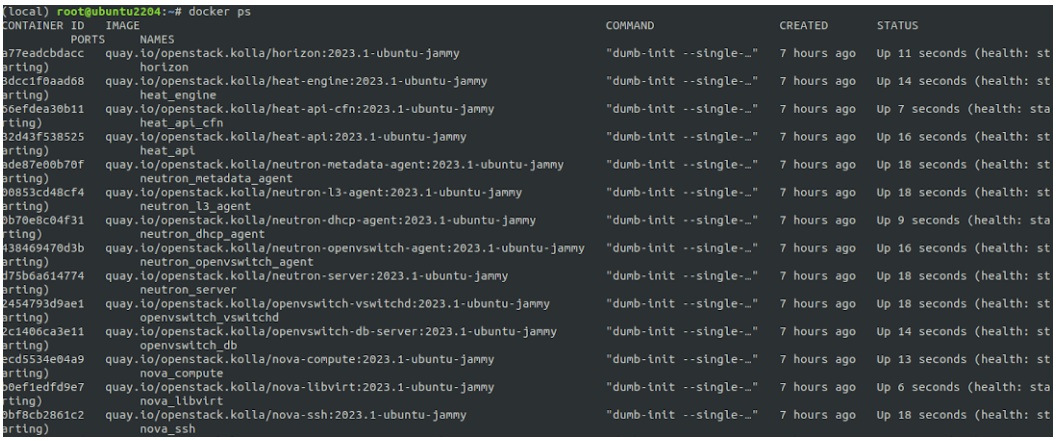


Figure 2.8 – A list of the running OpenStack service containers

Let's try to see OpenStack running in containers by firing a client test command. To fire a quick test, install the OpenStack client package:

```
# pip install python openstackclient
```

To interact with OpenStack API services, admin credentials are required, which can be generated by the `kolla-ansible post deploy` command, as follows:

```
# kolla-ansible post-deploy
```

Here's the result:

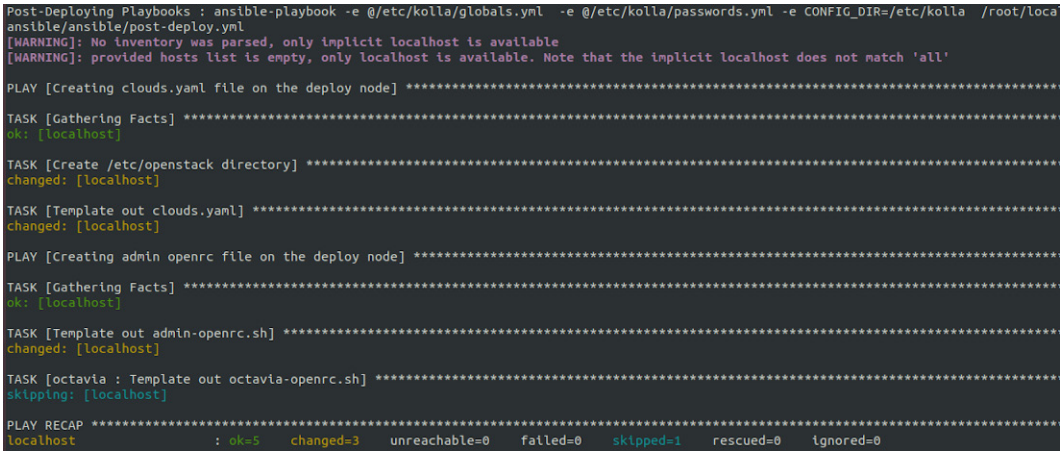


Figure 2.9 – The kolla-ansible OpenStack post-deployment output

The previous command will generate the `cloud.yaml` file, which contains different OpenStack admin credentials to interact with OpenStack services via the client command line. The content of the file variables can be populated by exporting the path of the generated `cloud.yaml` file, as follows:

```
# export OS_CLIENT_CONFIG_FILE=/etc/kolla/cloud.yaml
```

Validate the running OpenStack environment by listing, for example, the `nova` host:

```
# openstack hypervisor list
```

Here's the output:

```
(local) root@ubuntu2204:~# openstack host list
API has been deprecated. Please consider using 'hypervisor list' instead.
+------------------------+-----------+----------+
| Host Name              | Service   | Zone     |
+------------------------+-----------+----------+
| ubuntu2204.localdomain | scheduler | internal |
| ubuntu2204.localdomain | conductor | internal |
| ubuntu2204.localdomain | compute   | nova     |
+------------------------+-----------+----------+
```

Figure 2.10 – The OpenStack host list output

OpenStack services can be managed easily through the Docker container command-line interface. For example, Keystone container service logs can be viewed by running the following command line:

```
# docker logs keystone
```

The other option to inspect an OpenStack service container and report back container information, such as IP address and networking settings, can be performed by using the generic `docker inspect` command line, as follows:

```
# docker inspect keystone
```

Despite the flexibility of the usage of containers and the simplicity of destroying and building new ones in almost no time, in some cases, you might need to investigate a defined OpenStack service issue or configuration via shell access, as follows:

```
# docker exec --it nova /bin/bash
```

An all-in-one setup helps to develop and test code locally. Larger environments, including staging and production, would require an extended setup, which will be detailed in the next section.

Extending the deployment

Expanding on our initial physical design, discussed in *Chapter 1*, *Revisiting OpenStack – Design Considerations*, will require the evaluation of different hardware specifications and options for the chosen design. As we have narrowed down the methods of deployment and the different sets of tools

to build our first production iteration, we will start splitting the roles of OpenStack components across different physical systems. Unlike a testing environment, a production environment should follow a well-architected design pattern, based on the isolation of services, fault tolerance, scalability, and redundancy on each layer. We will deep-dive into the extension of those architectural aspects in *Chapter 3*, *OpenStack Control Plane – Shared Services*, and *Chapter 7*, *Running a Highly Available Cloud – Meeting the SLA*.

Our deployment pipeline will be the source of truth to deploy any environment if you consider going through a minimal all-in-one setup testing environment to a staging one, mimicking the production that will be promoted to production if all stages of the pipeline tests, including security ones, are executed with 0 errors.

> **Important note**
>
> Make sure to not introduce any user production workloads before getting a minimum level of redundancy in each control and data plane across your OpenStack environment. More details on extending to a highly available setup will be discussed in *Chapter 7*, *Running a Highly Available Cloud – Meeting the SLA*.

The deployment expansion for the environment should take an incremental approach by splitting the roles whenever feasible, and you should take into consideration that some services can be optionally separated on their own physical servers. A good practice is to start splitting additional roles in the physical layout once the core services are rolling and after gaining more operational expertise for each iteration. As the deployment pipeline feeds back the state of each new deployment, more technical decisions will need to be made to introduce improvements to the physical infrastructure layout, and you can choose which services can be moved to their own server in the next iteration. To safely start the deployment journey, we will need to install and configure our deployer before proceeding with infrastructure deployment.

Installing the deployer

Our deployer host will be installed with the following software and hardware settings:

- **Software**:
  - Jenkins version 2.414.1
  - Docker version 1.12.6 or greater
  - Ansible 2.3.1 or greater
  - Docker Python 1.10.0 or greater
  - Kolla-Ansible 16.1.0-16 – an OpenStack Antelope release or later
  - Git version control

    - **Operating system**: Ubuntu 22.04 LTS
    - pip and Python dependencies including `python-dev`
- **Hardware**:
    - **CPU**: At least 4 CPU cores and 64 bits x86
    - **Memory**: At least 4 GB of RAM
    - **Disk space**: At least 50 GB of free disk space
    - **Network**: At least two network interfaces

Start by installing Jenkins, by updating your local system and Java runtime:

```
$ sudo apt-get update
$ sudo apt install openjdk-11-jdk
```

Add the required GPG key for the Jenkins repository files:

```
$ curl -fsSL https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key
| sudo tee /usr/share/keyrings/jenkins-keyring.asc > /dev/null
```

The following command will address the local system repository to download a recent version of Jenkins:

```
$ echo deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc]
https://pkg.jenkins.io/debian-stable binary/ | sudo tee
/etc/apt/sources.list.d/jenkins.list > /dev/null
```

Run an update of the `apt` package list and install Jenkins:

```
$ sudo apt update
$ sudo apt install jenkins
```

Make sure Jenkins is running by firing the following command:

```
$ sudo systemctl status jenkins
```

If the Jenkins daemon is not running, issue the `start` command to launch its process:

```
$ sudo systemctl start jenkins
```

Once the Jenkins service is up and running, point to the CI/CD server address in a browser on the default `8080` port to access the user interface, and then proceed with the preliminary setup of the server:

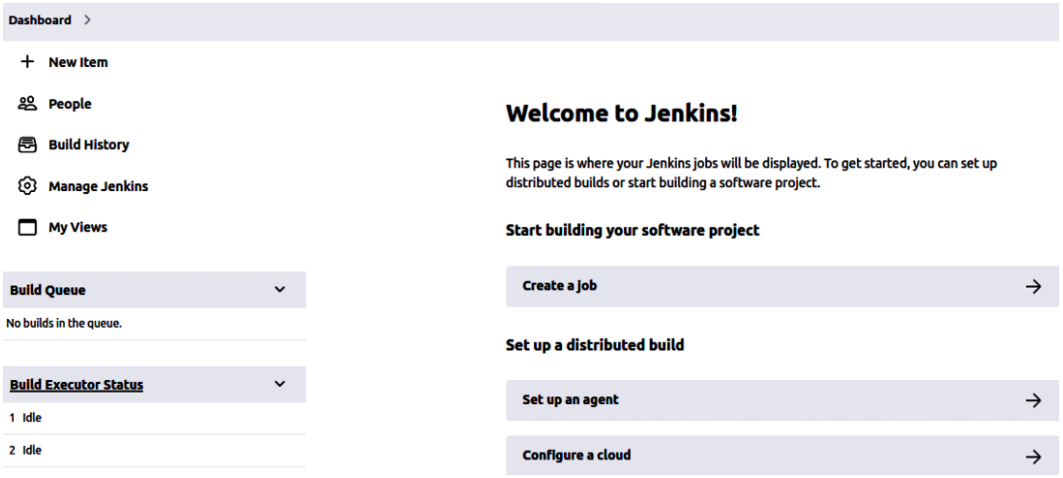


Figure 2.11 – Jenkins home page

First, we will need to add a few plugins before building the pipeline, starting with **Git SCM** plugins. To do this, click on **Manage Jenkins** on the upper-left corner tab. Type `git` in the search bar on the **Available Plugin** tab and click **Install without restart**:

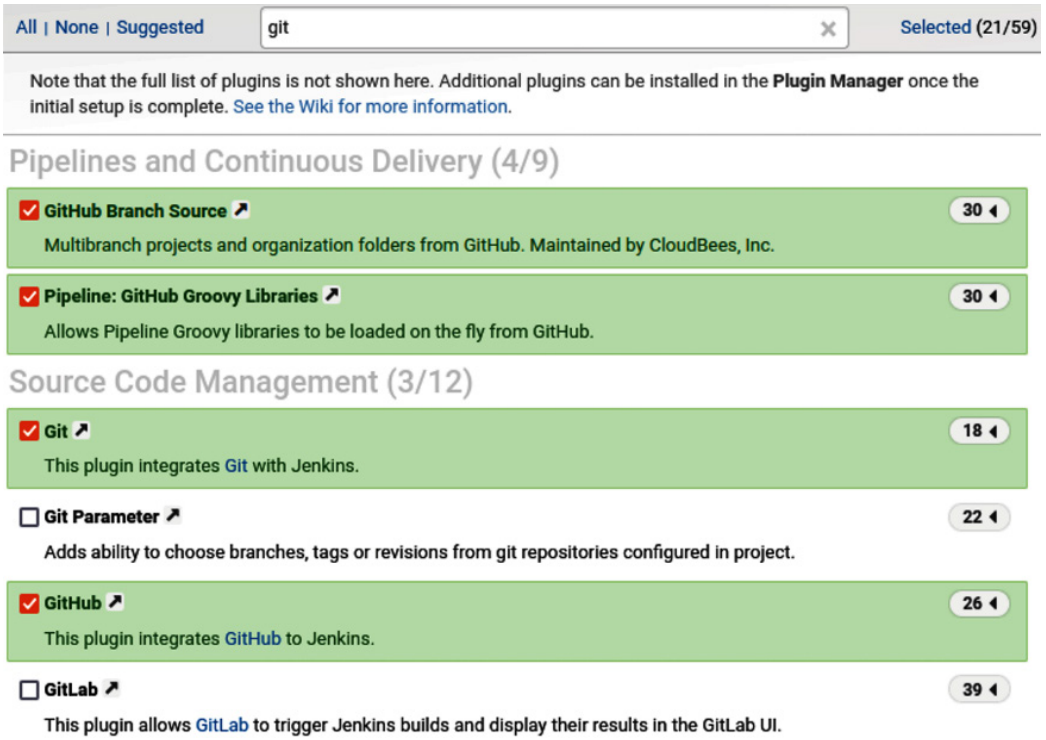


Figure 2.12 – Jenkins' Git SCM plugin setup

Once Jenkins and its required plugins are installed, we can move forward by setting up the CI/CD pipeline and different stages for future deployment.

Actioning the deployer

Our Jenkins file can be checked into the repository. The SCM pipeline will run the job definition with the configured path. The Jenkins pipeline definition can be summarized in high level in the following steps:

1. Create a local test environment on the destination testing host.
2. Check out the repository from a defined branch.
3. Install the required testing packages, including `kolla-ansible` and Docker runtimes.
4. Initialize the bootstrap server script.
5. Run Kolla prechecks on the bootstrapped testing environment.
6. Deploy the OpenStack testing environment.

> **Important note**
>
> Depending on the target environments you plan to run through the CI/CD pipeline, make sure to adjust the configuration layout of the main `globals.yml` and `inventory.yml` files separately in each environment. Each environment should run with a dedicated Jenkins pipeline file.

The next step is to create our first pipeline for the testing environment. Jenkins provides different options to create the job pipeline definition, either via the user interface wizard or by simply creating a Jenkins file that can live in the main directory of the OpenStack infrastructure as a code repository. For this purpose, our first job definition will follow the SCM method. The following snippets define the Jenkins deployment pipeline, starting with the first stage, where a virtual Python environment will be sourced to run each build in its own local environment:

```
pipeline {

  agent any

  stages {
    stage('Setup Local Environment') {
      steps {
        echo '--RUNNING LOCAL ENVIORNMENT --'
        sh ''' #!/bin/bash
        sudo apt-get update
        sudo apt-get install \
        python3-dev libffi-dev gcc libssl-dev docker.io -y
```

```
        sudo apt install python3-pip -y
        sudo apt install python3.10-venv -y
        sudo python3 -m venv local
        source local/bin/activate
        '''
      }
    }
```

The next stage will install all required dependencies, including `pip`, the Ansible runtime, and the `kolla-ansible` package:

```
    stage('INSTALLING PIP') {
      steps {
        echo '--INSTALLING PIP --'
        sh ''' #!/bin/bash
        pip install -U pip
        '''
      }
    }
    stage('INSTALLING Ansible') {
      steps {
        echo '--INSTALLING Ansible --'
        sh ''' #!/bin/bash
        pip install 'ansible-core>=2.16,<2.17.99'
        '''
      }
    }

    stage('INSTALLING Kolla Ansible') {
      steps {
        echo '--INSTALLING Kolla Ansible --'
        sh '''#!/bin/bash
        pip install \
        git+https://opendev.org/openstack/kolla-ansible@master
        kolla-ansible install-deps
        '''
      }
    }
```

Once all the required packages are installed locally, the next stage will define the file structure of the infrastructure code under the `/etc/kolla` directory and set a few parameters in the `globals.yml` file, such as the network interface and the type of distribution Kolla will use:

```
    stage('Preparing Infrastructure') {
      steps {
```

```
        echo '--Preparing Infrastructure Files Structure --'
        sh ''' #!/bin/bash
        sudo mkdir -p /etc/kolla
        sudo chown $USER:$USER /etc/kolla
        cp -r /usr/local/share/kolla-ansible/etc_examples/kolla/* \
        /etc/kolla/
        cp -r /usr/local/share/kolla-ansible/ansible/inventory/* \
        /etc/kolla/
        sed -i 's/^#kolla_base_distro:.ls*/kolla_base_distro: \
        "ubuntu"/g' /etc/kolla/globals.yml
        sed -i 's/^#enable_haproxy:.*/enable_haproxy: \
        "no"/g' /etc/ kolla/globals.yml
        sed -i 's/^#network_interface:.*/network_interface: \
        "eth0"/g' /etc/kolla/globals.yml
        sed -i 's/^#neutron_external_interface:.*/neutron_external\
        _interface: "eth1"/g' /etc/kolla/globals.yml
        sed -i 's/^#kolla_internal_vip_address:.*/kolla_internal\
        _vip_address: "10.0.2.15"/g' /etc/kolla/globals.yml
        '''
      }
    }
```

The next step will generate different secrets for all OpenStack services in their configurations. The secret file will inject each secret for each service during the OpenStack service installation:

```
    stage('Secrets Setup') {
      steps {
        echo '--Generating OpenStack Services Secrets --'
        sh '''#!/bin/bash
            kolla-genpwd -p /etc/kolla/passwords.yml
           '''
      }
    }
```

Next, the pipeline will instruct Jenkins to prepare the container scripts for booting, including the system configuration:

```
    stage('Boostrap Servers') {
      steps {
        echo '--Running Ansible Kolla Boostrap Server Script --'
        sh '''#!/bin/bash
             kolla-ansible -i /etc/kolla/all-in-one bootstrap-servers
           '''
      }
    }
```

The next stage will run a precheck task to make sure that the bootstrap step was performed without issues:

```
    stage('Infrastructure Pre-Checks') {
      steps {
        echo '--Running Ansible Kolla Prechecks Script --'
        sh '''#!/bin/bash
            kolla-ansible -i /etc/kolla/all-in-one prechecks
           '''
      }
    }
```

The final stage is to deploy the services, pull the images, and run the services in containers:

```
    stage('Deploy Infrastructure') {
      steps {
        echo '--Running Ansible Kolla Prechecks Script --'
        sh '''#!/bin/bash
        kolla-ansible -i /etc/kolla/all-in-one deploy
        '''
      }
    }
  }
}
```

Make sure to have a new testing branch:

```
$ git branch test
```

Commit and push the created file to the source code repository with the new branch:

```
$ git add Jenkinsfile
$ git commit -m 'Initial Test Environment Pipeline'
$ git push --set-upstream origin test
```

> **Important note**
>
> Using the Jenkins file the SCM way enables us to treat the pipeline definition as code. Make sure that all changes and updates for any job definition go through the pipeline code. The pipeline file is written in Groovy syntax. A recommendation is to use an IDE to highlight the Jenkins file code format in the same way as for YAML. If your IDE does not highlight the Groovy syntax, add `#!/usr/bin/env groovy` to the top of the pipeline Jenkins file.

From the Jenkins user interface, select **New Item** from the top-left menu. Select **Pipeline** as the type of job, with the desired name to run the test pipeline. The next wizard creates a pipeline item, named `openstack-dev`:

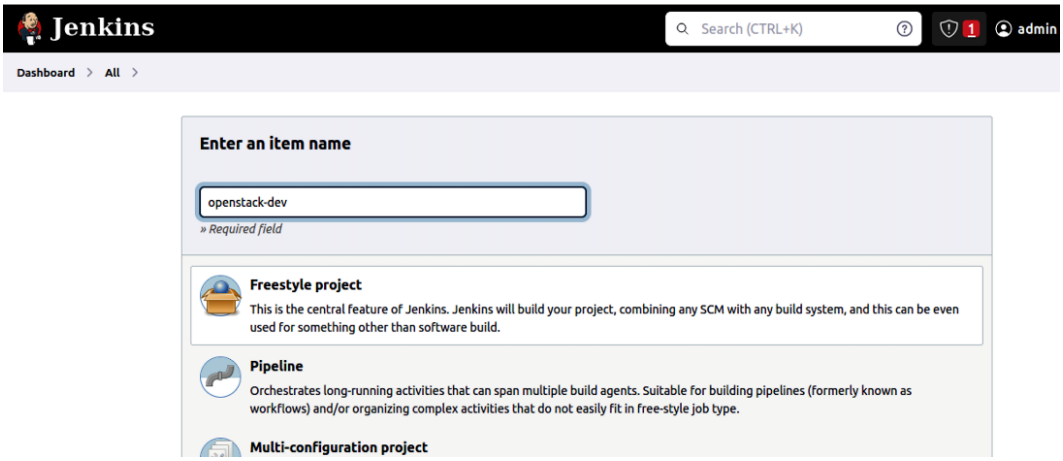


Figure 2.13 – Jenkins pipeline creation

Select the pipeline script from SCM in the **Configure** tab and Git by specifying the repository URL. Make sure to configure the script of the Jenkins file in the repository directory path:

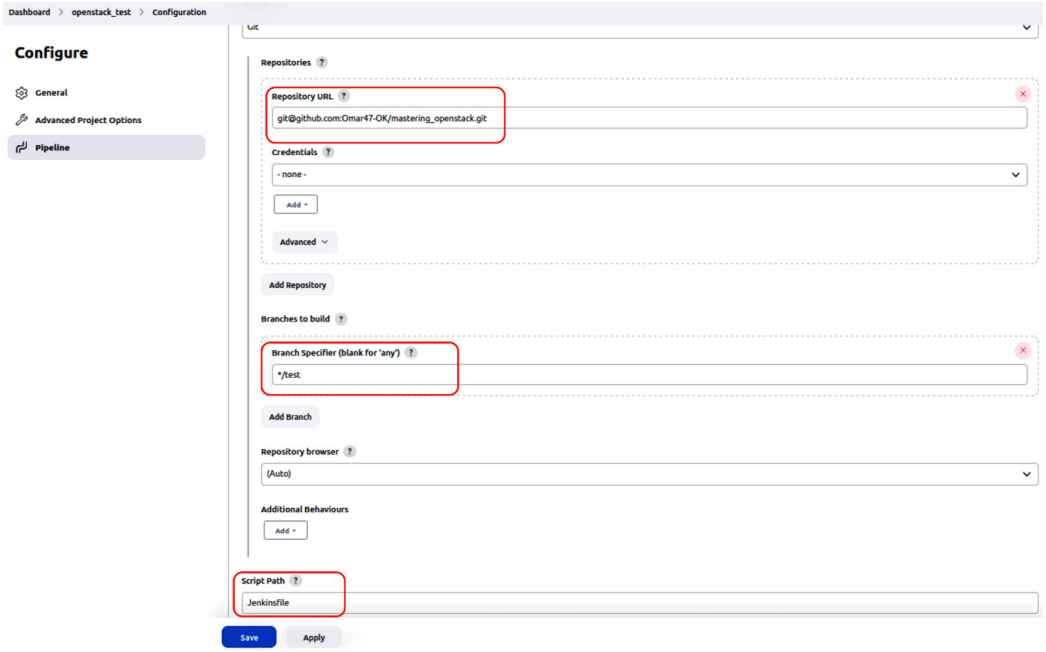


Figure 2.14 – The Jenkins pipeline SCM job definition

Our first deployment setup for the testing environment is ready to go. All that is left is to push the button. Click on **Build Now** on the left-hand side of the newly created job definition in Jenkins, where the different stages declared in the pipeline code will be visible:

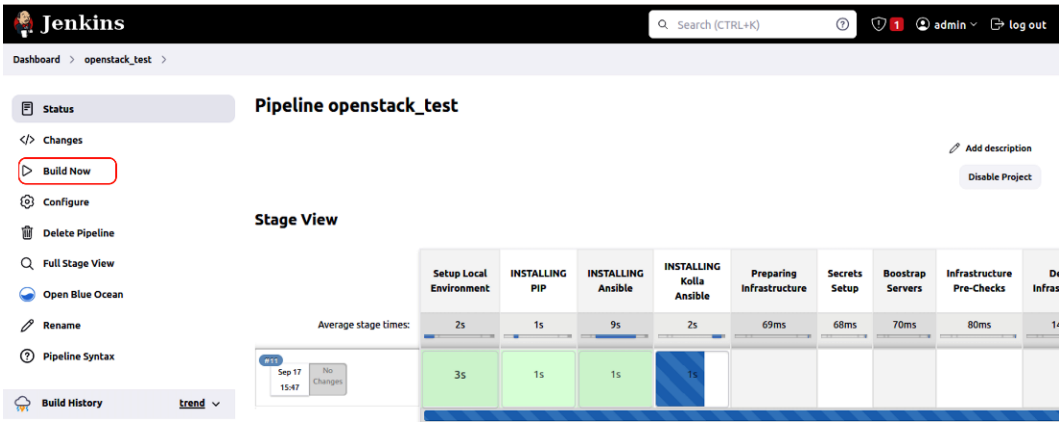


Figure 2.15 – The Jenkins OpenStack test deployment pipeline

Jenkins jobs that are running can be monitored by clicking on the **Console Output** tab, where each stage defined in the pipeline file is reported to support further debugging and troubleshooting when possible errors occur:

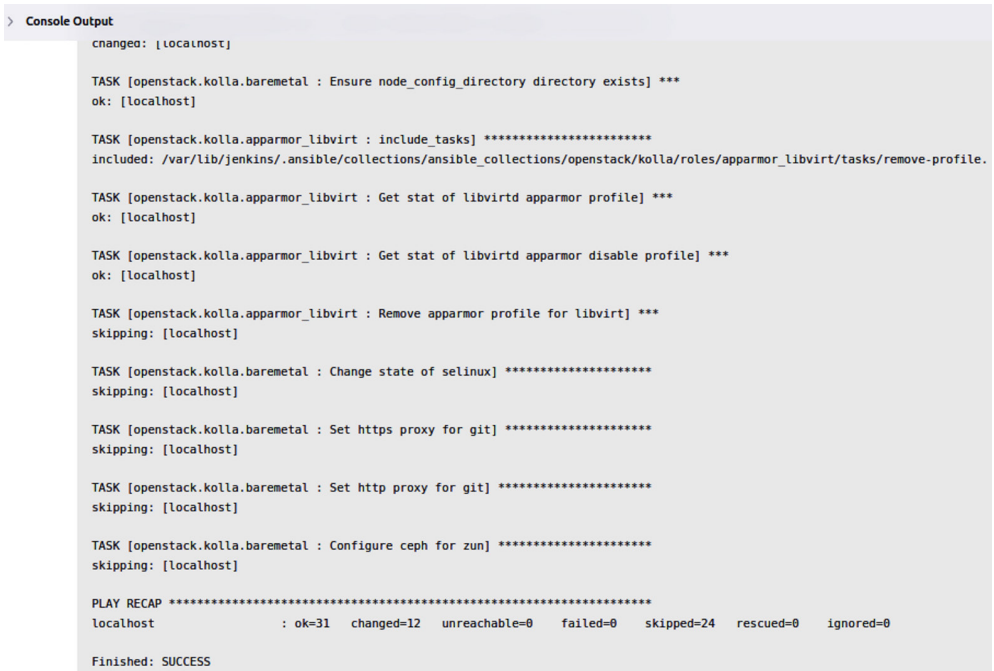


Figure 2.16 – Jenkins OpenStack test deployment output

The Jenkins console output is very handy for troubleshooting any possible issues that could occur during the Ansible run. Make sure the `failed` status is `0`. When the deployment status ends with the `SUCCESS` value, we can enforce the pipeline with additional security checks.

Deploying with security

As we emphasized at the start of this chapter, security should be considered in the early steps of building and deployment projects. That is what has been defined as a *shift to the left*. The craft of automating security checks should be learned in testing environments to report any potential vulnerabilities before promoting the infrastructure to a production environment.

Our first security enforcement will be conducted by installing an additional Jenkins plugin, which will be an interesting asset to our DevSecOps pipeline. As we are dealing with containers as the main artifacts, we will install a container inspection plugin to perform security scans on the built containers at a later stage. **Anchore** is one of the most widely used container vulnerability-scanning platforms, providing seamless integration with CI/CD tools, source code, and even container registries for both Docker and Kubernetes. The Anchore plugin exists for Jenkins and can be installed straightforwardly in the same way we installed the Git plugin. On the **Available plugins** tab, search for `anchore`:

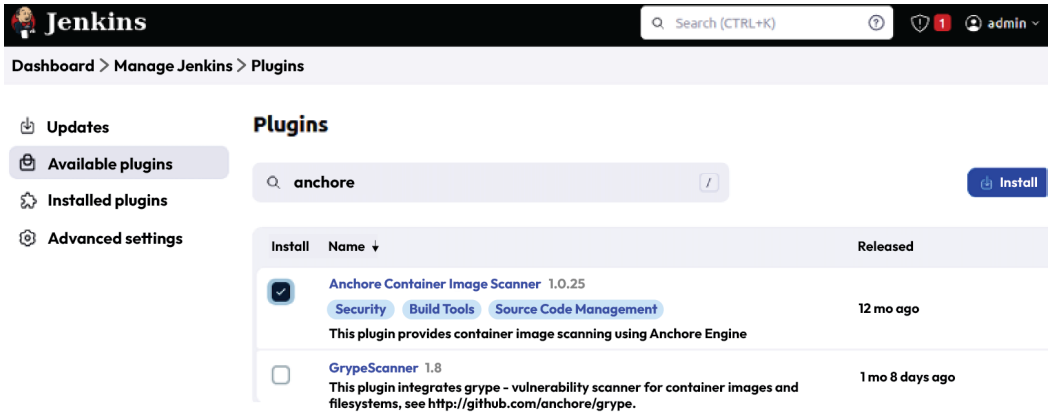


Figure 2.17 – Searching for the Jenkins Anchore plugin

Select the **Anchore Container Image Scanner** plugin and click on **Install without restart**:

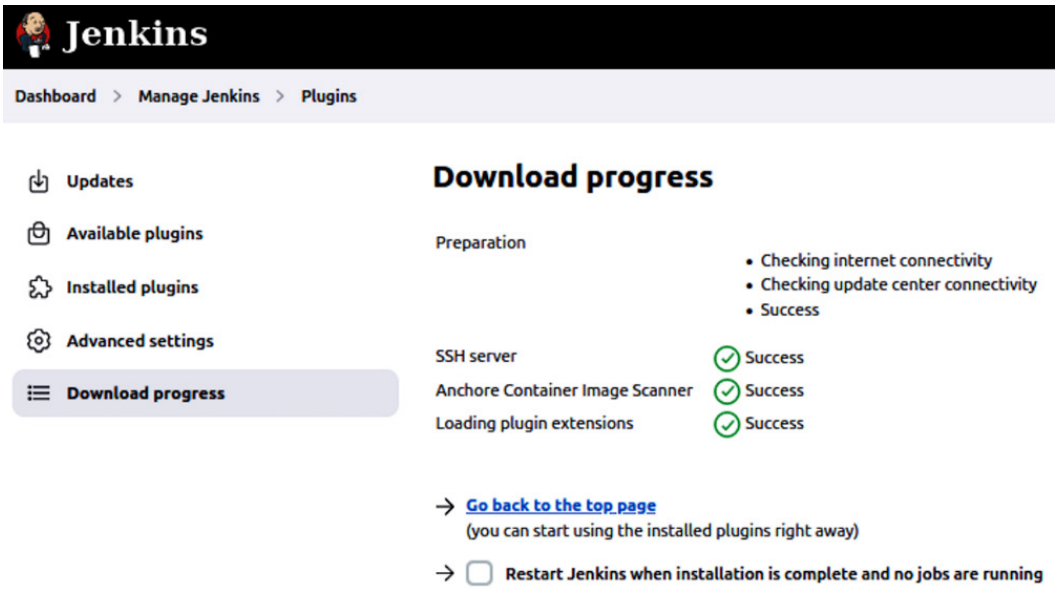


Figure 2.18 – The Jenkins Anchore plugin setup

Important note

More details about the Anchore platform can be found here: `https://anchore.com/container-vulnerability-scanning/`.

Once the plugin is installed, we need to adjust our settings to instruct our deployer to use a Docker repository at a specific address and port. For this example, the image registry will be deployed in the deployer host, named `master_registry`, by running the following command:

```
$ sudo python3 -m pip install kolla
$  docker run -d \
   --network host \
   --name master_registry \
   --restart=always \
   -e REGISTRY_HTTP_ADDR=0.0.0.0:4000 \
   -v registry:/var/lib/registry \
   registry:2
```

Make sure to update the `globals.yml` file to use the local Docker registry by editing the following configuration lines:

```
…
docker_registry: 10.0.2.15:4000
docker_registry_insecure: yes
...
```

Commit and push your changes to the main infrastructure code repository so that our CI/CD will point to the local registry in later image build steps:

```
$ git commit -m 'Docker registry global setting'
$ git push
```

Before getting the image build pipeline running, we will install the Anchore engine to run the container inspection later. The Anchore open source version comes as a Docker image that can run standalone. Proceed by downloading the Anchore engine `docker-compose` file, as follows:

```
$ curl https://engine.anchore.io/docs/quickstart/docker-compose.yaml >
docker-compose.yaml
```

Optionally, reset the admin password defined by the `ANCHORE_ADMIN_PASSWORD` configuration line in the `docker-compose.yaml` file. That will be needed for later steps.

> **Important note**
>
> For permanent access to the Anchore engine interface, export the Anchore endpoint, username, and password to the `ANCHORE_CLI_URL`, `ANCHORE_CLI_USER`, and `ANCHORE_CLI_PASS` environment variables, respectively. If the variables are not populated within the operating system, you will need to export them in each run of the CI/CD pipeline execution.

Now, let's create and run the Anchore engine Docker container:

```
$ sudo apt install docker-compose
$ docker-compose up –d
```

The Anchore CLI is very handy to interact with the engine and can be installed as follows:

```
$ git clone https://github.com/anchore/anchore-cli
$ cd anchore-cli
$ pip install --user --upgrade .
```

On the Jenkins dashboard, configure the installed Anchore plugin by pointing to **Manage Jenkins** and clicking on **System**. Provide the engine username of `admin`, the password, and the engine URL defined in the `docker-compose.yaml` file for the Anchore engine:

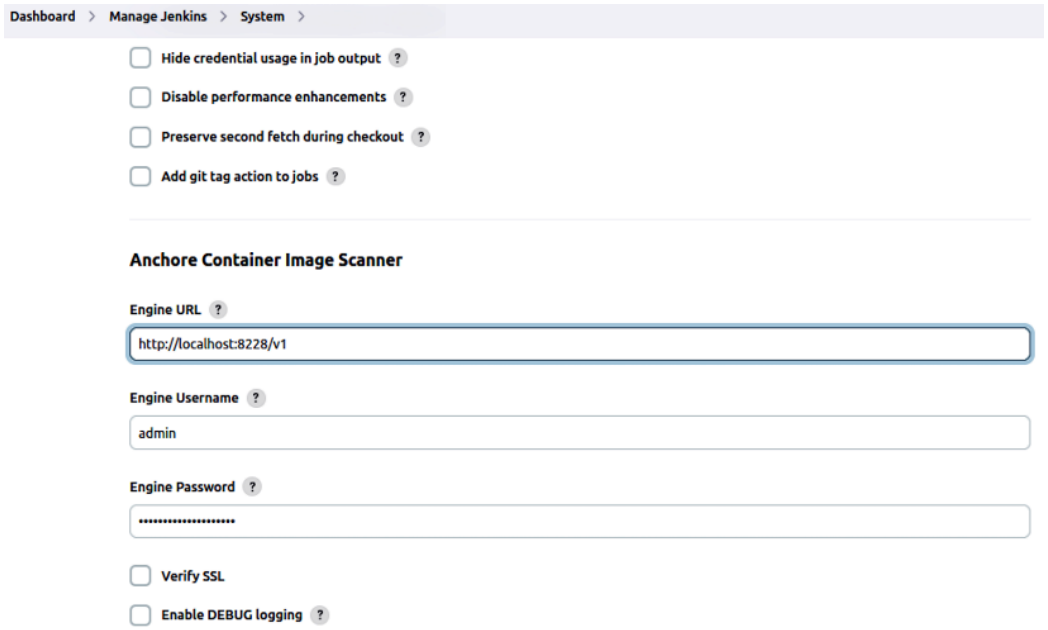


Figure 2.19 – The Jenkins Anchore engine configuration

Important note

The Anchore engine open source project is not maintained anymore but is still available on GitHub and Docker Hub for evaluation, along with its Jenkins plugin. An enterprise edition has replaced the open source version. More info on the open source Anchore engine can be found here: `https://github.com/anchore/anchore-engine`. Other similar container security inspection solutions can be found at `https://github.com/anchore/syft` and `https://github.com/anchore/grype`.

Building images and pushing them to the local registry can be automated by optionally using a dedicated pipeline, with the following steps:

1. Build an OpenStack service Docker image using Kolla.
2. Run an Anchore security check on the pushed image.
3. Push the created OpenStack service Docker image using Kolla if no specific vulnerabilities have been detected by Anchore.

Our image pipeline script is structured in three stages – build images, Anchore security scans, and push images – if the vulnerability scans are successful. This can be depicted as follows:

```
pipeline {
   agent any
   stages {
       stage('Build Images') {
           steps {
               echo '-- KOLLA BUILD OPENSTACK SERVICES IMAGES --'
               sh '''#!/bin/bash
               kolla-build -b ubuntu
               '''
           }
       }

       stage('Anchore Security Scan') {
           steps {
               echo '--RUNNING LOCAL ENVIRONMENT --'
               sh '''#!/bin/bash
               anchore-cli registry add 10.0.2.15:4000
               anchore-cli image add master_registry:latest
               '''
           }
       }

       stage('Push Images') {
           steps {
               echo '--RUNNING LOCAL ENVIRONMENT --'
               sh '''#!/bin/bash
               kolla-build --registry 10.0.2.15:4000 --push
               '''
           }
       }
   }
}
```

Save the pipeline script in Jenkins and run the pipeline from the new job definition:

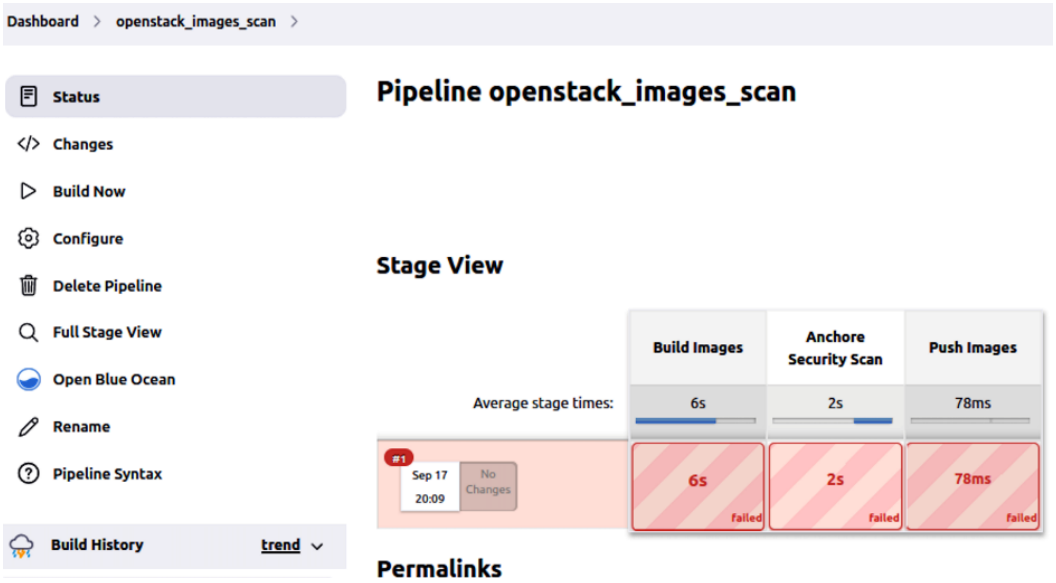


Figure 2.20 – Running the OpenStack inspection image pipeline

Running the OpenStack security scan image pipeline will generate a useful vulnerability assessment and trigger a `stop` action so that the pipeline will fail. Anchore reports the cause of the failure from the previous run, which did not comply with the engine policy.

Automating the OpenStack deployment by starting with a development environment helps to deliver a higher code quality empowered by security guardrails. Cloud operators will learn about possible issues during the cloud infrastructure deployment and master how to solve them earlier. It goes hand in hand with security and anomaly detection practices, which should support the move to production with more confidence.

Summary

This chapter has been a quite milestone in defining the right way and best practices to run a complete OpenStack deployment in a DevSecOps style. Inheriting modern, agile software development practices in infrastructure management is considered a great opportunity for cloud operators, and hence for DevSecOps, to focus on infrastructure improvement and extension, rather than being blocked by traditional, manual operations tasks. The OpenStack ecosystem, with the latest additional services and functions, can be cumbersome to manage if no proper agile methods are considered from the start of the journey. This chapter has highlighted security fusion from day one. Silos should no longer exist between infrastructure and security duties. Undoubtedly, that is a game changer toward a successful deployment experience, armed with security and automation. As we have built a solid draft together with the right tooling and processes, we will take our momentum to the next chapter to extend our production design and deployment, cover OpenStack clustering, define its various services' roles, and deploy them in a DevSecOps way.

3

OpenStack Control Plane – Shared Services

"The value of an idea lies in the using of it."

– Thomas A. Edison

Addressing the challenge of business continuity within the OpenStack private cloud comes down to the traits and capabilities of its control plane. As highlighted in the first chapter, a variety of core services, including compute, imaging, storage, networking, and identity, should be designed for deployment with scalability and security in mind at an early stage. The other facet of such a requirement is to treat other non-native OpenStack services the same way as core ones by designing for failure and scalability. Such services include messaging queues and databases. As we consider best practices for deploying to production, it is essential to iterate through each service that would be part of the OpenStack control plane. Another challenge that arises during the deployment is the way OpenStack services are built and deployed. As we saw in *Chapter 2*, *Kicking Off the OpenStack Setup – The Right Way (DevSecOps)*, containers to run our OpenStack services, breaking the control plane services into smaller chunks of units, will increase the reliability and scalability of each running service. In this chapter, we will extend our designed layout to a production-ready environment by going through the following topics:

- Defining the perimeter of the OpenStack control plane
- Identifying different OpenStack services taking up part of the control plane
- Preparing a first production deployment
- Deploying the next OpenStack environment using kolla-ansible
- Uncovering additional custom options for the OpenStack control plane deployment

The OpenStack control plane

When we started designing our first iteration layout, we encountered a variety of OpenStack services, each composed of several components. One or several components can be designed to run in separate physical machines assigned with roles for logical operations and management reasons. As highlighted in *Chapter 1*, *Revisiting OpenStack – Design Considerations*, separating the roles logically will support the orchestration of the OpenStack deployment and operation. The term *control plane* is widely used in the network glossary. It dictates how to process and run requests. As an abstract concept, the control plane can be formed of one or many system components that centrally manage and describe the way a request can be fulfilled. When referring to the OpenStack logical layout, a cloud controller is designated to run most of the pieces of the control plane.

Running the control plane – cloud controller

The OpenStack APIs, scheduler services (including compute, storage, and file share schedulers), API endpoints, databases, and messaging queues are counted as part of the OpenStack control plane. This section iterates through each service that is part of the cloud controller setup and summarizes them into two main categories, OpenStack and infrastructure services, as follows:

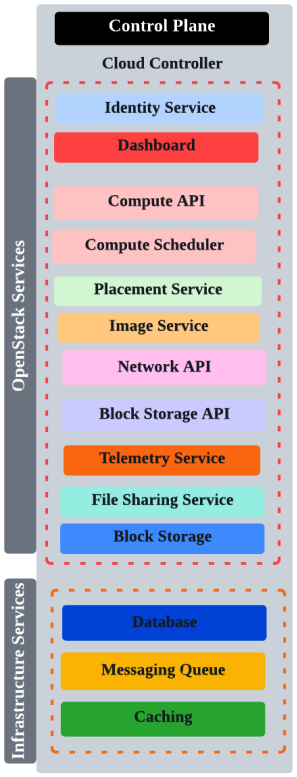


Figure 3.1 – An overview of the OpenStack control plane

We will walk through the control plane in the following subsections and examine each service, its subcomponents, and the latest updates.

Identity service

The identity service, as explored in *Chapter 1*, *Revisiting OpenStack – Design Considerations*, presents a vital component to authorize and authenticate service communication to accomplish a specific request, such as storage creation or instance provisioning. As part of the control plane, Keystone consolidates a catalog of all existing OpenStack services to be reached through its API.

It is important to provide a glossary of key terms of Keystone when running the control plane services. This is to ensure compliance and governance policies when dealing with different sets of domains, projects, and users as follows:

- **Project**: The concept of a *project* has replaced that of a *tenant* within version 3 of the Identity API. The concept did not change, as a project is a logical construction of a set of resources isolating users and groups.
- **Domain**: Going up a layer, a domain contains a set of users, groups, and projects. This abstract isolation could be very helpful for large enterprises with different sets of departments, for example. Each department can be allocated a domain made up of different projects and users.
- **Role**: One of the key mechanisms of the Keystone service since its early days is the concept of the *role*. A user or group of users can be assigned to different projects with different authorization layouts without the need to create multiple users with different assignments and switch manually to a different environment to access specific resources through different projects.
- **Catalog**: Keystone grants access to different services through its discovery service. Those services, from a Keystone perspective, are exposed as OpenStack service endpoints and registered in the service catalog.
- **Token**: Accessing a specific service must be done through obtaining a token. It can be considered as an information card that validates the user, as well as the expiration date and time. It also checks which project(s) the token has been associated with, and to which endpoints. The token concept itself has passed through different versions and ways of working. When expanding a production OpenStack environment, it is necessary to have a basic understanding of the scoping mechanism for the Keystone token. As a large number of users will be competing to access services and resources, scoping will help to better organize the access boundaries in a Keystone request, which can be categorized into two groups:
    - **Scoped token**: Once a token request is created, the Identity API will specify which OpenStack services (endpoints) will be used by the requester and its associated roles. It mainly contains information such as the reference to the project and domain ID.

    - **Unscoped token**: Unlike the scoped token, the created token request will not contain any specific authorization scope to access services. When users present credentials for the first time, the usage of unscoped tokens avoids an excessive loop of authentication flow. At a later time, the unscoped token can be narrowed down to a scoped one to limit authorization to a specific service component(s).

- **User**: This is an API requester to consume OpenStack services and resources.
- **Group**: This is a collection of users or API requesters belonging to the same domain, as explained previously.

The Keystone API has been improved by providing more options for authentication mechanisms aside from the traditional password way, such as SAML2 and OpenID Connect based on OAuth. Identity federation support has been a big milestone for the Keystone implementation, enabling pluggable **Identity Providers** (**IdP**) through different backends, including SQL, LDAP, and even a federated IdP. Within the latest OpenStack releases, including Antelope and Bobcat, several backends have been supported, allowing enterprises to leverage their existing authentication/authorization systems for more consolidated management.

A brief summary of the current supported authentication backends is as follows:

- **LDAP**: Many organizations might have existing LDAP instances and require a set of security policies to be followed for different compliance reasons. In this setup, Keystone will need to grant access to the LDAP servers to read and write and will not handle an IdP role internally.
- **SQL**: Most of the default OpenStack deployments come with SQL configuration to store user information in a SQL database and provide direct interaction with the Keystone service.
- **Federated IdP**: This is one of the most common setups, allowing organizations to leverage a complete consolidated identity point where Keystone will establish a trust relationship and act as a service provider. The OpenStack community has developed more stable releases around this layout, offering more ways of authentication against multiple backends, such as SQL, LDAP, and Active Directory, through SAML assertions. Lately, there has been more work on OpenID Connect and OAuth in its second version. This even allows authentication using public social login platforms such as Google and Facebook.

Important note

The Keystone Identity API supports the full **Create, Read, Update, and Delete** (**CRUD**) RESTful operations API. Commands are fired using the OpenStack CLI and not the Keystone CLI as the latter is no longer supported within Antelope and later releases, and the latest OpenStack releases that have adopted Identity API version 3.

Depending on which authentication and authorization strategy you adopt, the mechanism of Keystone is the same when dealing with pure OpenStack ecosystem workflows. The Identity API's interaction with the rest of the services might be overwhelming to tackle when dealing with a large infrastructure layout. At this point, once we have iterated through the latest state of the art in Keystone, a brief overview of how interactions work will be helpful before listing the next services. We should remember that Keystone will turn out to be the security hub of AA in the OpenStack ecosystem where default configurations would not be ideal to avoid possible security gaps and further boost your compliance and governance.

> **Important note**
>
> Keystone configuration can be customized through the Keystone `kolla-ansible` playbook located at `/kolla-ansible/ansible/roles/keystone`. At the time of writing this book, the default `kolla-ansible` configuration within the Antelope and later releases uses Keystone fernet tokens to authenticate uses instead of traditional **Public Key Infrastructure** (**PKI**)-based tokens.

The following workflow briefly demonstrates typical Keystone request and response interactions between an API requester (user or system) and identity service to access a specific OpenStack service:

1. A user or API requester authenticates against Keystone service using credentials. Identity API receives the request and grants a token to the requester once authenticated.
2. The user or API requester presents the request with the received token to access the demanded service.
3. The demanded service validates the eligibility of the presented token with the identity service.
4. Once validated, the requested service replies with a response to the API requester.

This workflow is depicted in the following figure:

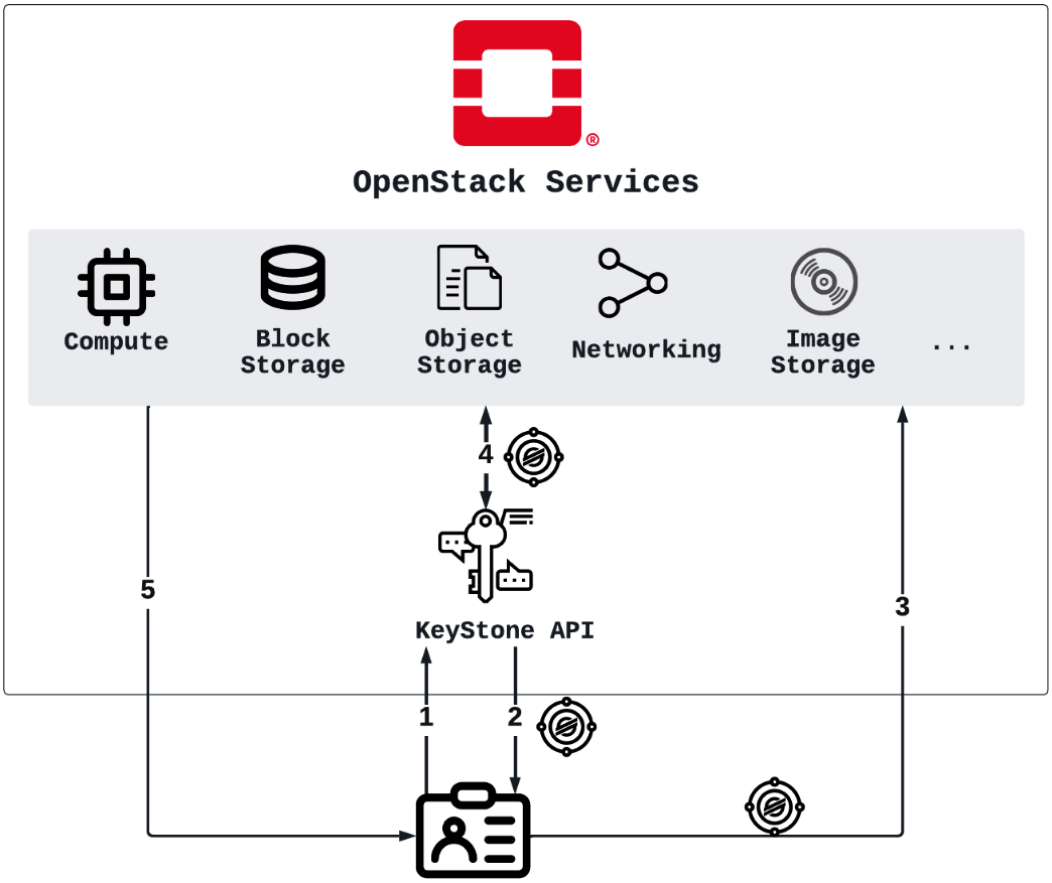


Figure 3.2 – Keystone API request processing

As noted previously, any request to access a specific service or resource will engage the identity service and later interactions will be handled by API requests. Taking a more complex scenario, such as virtual machine creation, more interactions will be involved. This would engage more services to be contacted, such as image, network, and storage, forming the essential resources to run an instance. Hence, Keystone will validate each request by using the original authentication token. A new desired service request will require a newly generated authenticated request using the first authentication token.

Important note

Keystone now has OAuth 2.0 Mutual TLS support. Keystone Auth has a new plugin for the OAuth 2.0 Device Authorization Grant.

Compute service

Our control plane includes all compute service components except `nova-compute`, which will be running in dedicated hosts. As covered in *Chapter 1*, *Revisiting OpenStack – Design Considerations*, the composition of different compute daemons constructs the main capability of the OpenStack ecosystem. The following Nova services will run as part of the cloud controller stack:

- `nova-api`: This is the primary interface to handle compute requests, including the creation, listing, deletion, and management of compute resources.
- `nova-scheduler`: This is a set of filters and weights to determine the most suitable compute hosts to run an instance based on custom or default characteristics such as host location, memory, or CPU architecture.
- `nova-conductor`: This is considered a novel addition to the Nova architecture prior to the latest OpenStack release, `nova-conductor` insulates the compute process from contacting the database directly (counted as not trusted from a security perspective) and connects the database directly instead.
- `nova-novncproxy`: This provides VNC console access to the instances.

> **Important note**
>
> The Nova playbook is located at `/kolla-ansible/ansible/roles/nova` in the `kolla-ansible` repository. Note that the playbook does not separate each compute component on its own.

Scheduling service

The Placement service, as highlighted in *Chapter 1*, *Revisiting OpenStack – Design Consideration*, is considered a great addition to the compute service to conduct fine-grained prefiltering and achieve advanced instance scheduling not only by checking compute resources but also for networking and storage pools. The Placement service was introduced as part of the Nova architecture, and due to its evolution, it has been turned into a separate project. Some important constructs that use this service are as follows:

- **Resource providers**: The Placement service tracks the underlying resources (types and occurrences), such as compute, storage pools, and networking, in the form of abstracted data model objects.

- **Resource class**: Tracked resources are classified by types that are categorized into two main sets of resource classes:
  - **Standard**: `MEMORY_MB`, `VCPU`, `DISK_GB`, `IPV4_ADDRESS`, `PCI_DEVICE`, `SRIOV_NET_VF`, `NUMA_CORE`, `NUMA_SOCKET`, `VGPU`, and so on.
  - **Custom**: Starting with the `CUSTOM_` prefix, custom resources can be created and added to the resource classes of the Placement service.
- **Inventories**: A collection of a set of resource classes provided by a given resource provider. For example, a resource provider, `RP01`, is associated with four resource classes: `16 VCPU`, `4096 MEMORY_MB`, `200 DISK_GB`, and `50 IPV4_ADDRESS`.
- **Traits**: These represent the characteristics of a resource provider. For example, vCPU provided by a resource provider, `RP01`, can be `HW_CPU_X86_SVM` (SVM x86 hardware architecture) and `DISK_GB` can be **Solid-State Drive** (**SSD**). Both traits assigned to `RP01` will be noted as `is_X86_SVM` and `is_SSD` for both `VPCU` and `DISK_GB` resources, respectively.

> **Important note**
>
> The Placement service provides standard traits; however, operators can still create custom traits by using the placement API CLI. Traits can be specified in the Glance image characteristics, or the extra specs of the instance flavor used for the instance launch.

- **Allocation**: An instance that occupies resources in a compute node is considered a consumer for a resource provider. This occupation of resources is represented as *allocation* stored in a data model record.
- **Allocation candidates**: After processing each request, the Placement service reshuffles the existing resource providers and returns the latest available group to be available for the next suitable requests.

Through the Placement service, more advanced filtering options can be achieved. The following workflow demonstrates how the networking service interacts with the compute service through placement. The desired scenario is to schedule instance provisioning in a compute node based on the network bandwidth:

1. An inventory is created through the compute service.
2. A resource provider is created by the compute node.
3. Under the compute resource provider, a networking resource provider is created by the networking service.

4. The networking service reports the bandwidth inventories.
5. The networking service provides a resource request for a port.
6. The compute service receives the resource request from the port and adds it to the API request in a `GET /allocation_candidates` request.
7. The exposed candidates will be filtered by the Placement service and the compute service will pick up the most suitable one.
8. The compute service claims the resources in the Placement service.
9. The compute service exchanges the allocation update with the networking service.
10. The Placement service updates its allocation candidates for the next request.

The placement API provides additional capabilities by assisting the Nova scheduling process. Prior to the Newton release, only computing resource usage (such as CPU and RAM) was tracked and managed. With the addition of multiple resource providers (storage and network services), the placement API simplifies resource recording and scheduling through a unified management method.

Image service

The Glance API is part of the control plane. However, as was highlighted in *Chapter 1*, *Revisiting OpenStack – Design Consideration*, storing the images can be handled by different data store types, such as Swift object storage, AWS S3, the Ceph storage system, VMware, filesystems, and, recently, the Cinder block storage service. The evolution of the Glance service has been marked mostly by the flexibility of the storage image backend support within the latest OpenStack releases. Within Antelope and later releases, the Glance API service can be configured to support a list of backends at the same time, where the Glance API will process the disk image operation using the default data store; otherwise, it will go through the defined backend list.

> **Important note**
>
> By default, the imaging service playbook supports file, HTTP, VMware, Cinder, Swift, and Ceph backends located in the `/kolla-ansible/ansible/roles/glance/defaults/main.yml` file. Enabling or disabling additional image storage backends can be done from the `/kolla-ansible/etc/kolla/globals.yml` file by setting `yes` or `no` to `glance_backend_ceph`, `glance_backend_file`, `glance_backend_swift`, and `glance_backend_vmware` for Ceph, file, Swift and VMware storage backends, respectively.

The `glance-registry` process running in the control plane will handle the metadata updates for each associated image in the database, including the image location defined in the storage backend path, as illustrated in the following figure:

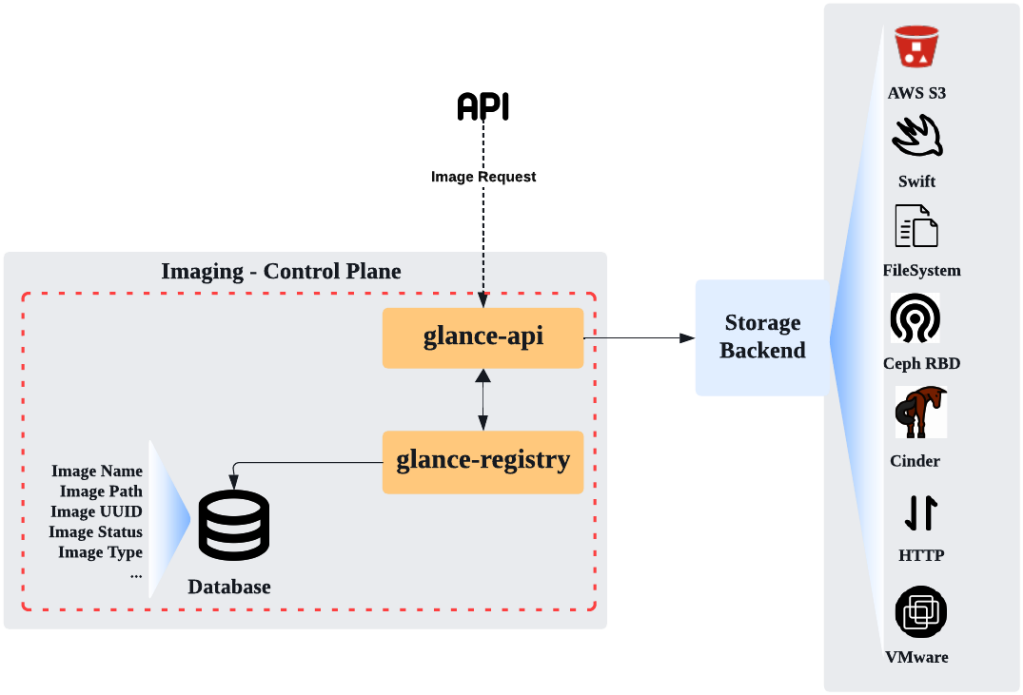


Figure 3.3 – The Glance control plane components

The other great addition to this Glance storage support capability is that the control plane can be customized to support multiple block storage backends for Glance. If two Cinder storage volumes are created, a property can be assigned to a volume, for example, SSD or HDD, where the Glance API will be instructed to use the desired Cinder volume based on the cloud operator's desired configuration.

Networking service

Unlike many other OpenStack services, the Neutron service is considered more complex due to the nature of its evolution through the OpenStack releases as well as its advanced virtual networking features and extensions. As demonstrated in *Chapter 1*, *Revisiting OpenStack – Design Consideration*, the networking capacity and segmentation should be planned in advance not only to accommodate more circulating traffic but also to take advantage of virtual networking scalability. For this, a network node is dedicated to handling different types of tenants and providers by design. Meanwhile, the main Neutron API will run on the controller node as part of the control plane. The number of Neutron drivers and plugins is increasing with each OpenStack release; within the Antelope release, most of the well-known SDN driver mechanisms are supported, including `openvswitch`, `OVN`, `linux bridge`, `l2population`, and `baremetal`. Advanced service plugins providing virtual network features such as VPNs, firewalling, and routing run on a separate network node. It is strongly recommended to consider hardware network bonding for both cloud controllers and network nodes for performance and resiliency reasons.

Important note

Networking configuration through the `kolla-ansible` infrastructure code repository can take different forms depending on the networking features to be added. All mechanisms, drivers, and service plugins are listed in the Neutron file (`/kolla-ansible/ansible/roles/neutron/defaults/main.yml`). Neutron drivers such as Open vSwitch and OVN run on their own playbooks located in the same folder path, `/kolla-ansible/ansible/roles/openvswitch` and `/kolla-ansible/ansible/roles/ovn-controller`, respectively

Block storage service

Cinder's main mission since its incubation within the OpenStack ecosystem has been to provide persistent storage to instances. Unlike ephemeral storage, the block storage concept has been included in OpenStack since Bexar's early release and kept as part of the Nova project. However, since the Folsom release, it has been changed to its own service. There are many reasons for this separation; the main reason is the development of the Cinder architecture. Some of the block storage subcomponents are considered part of a control plane, including `cinder-api` and `cinder-scheduler`, as illustrated in the following figure:

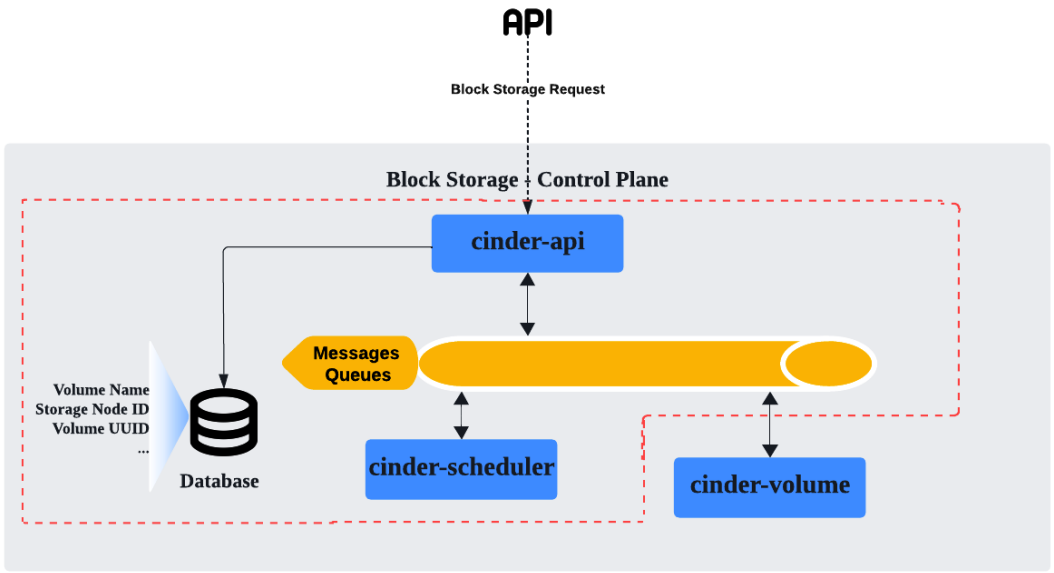


Figure 3.4 – Cinder control plane components

We will cover the block storage extensions in more depth in *Chapter 5*, *OpenStack Storage – Block, Object, and File Shares*.

Important note

Block storage is not enabled by default in the current `kolla-ansible` infrastructure code repository. Setting `enable_cinder` to `'yes'` in the `/kolla-ansible/etc/kolla/globals.yml` file will enable created instances to store their disks in the associated volume. Supported Cinder storage backends are listed in the Cinder playbook defined in the `/kolla-ansible/ansible/roles/cinder/default/main.yml` file and referenced with the `cinder_backend` configuration stanza code. Make sure to prepare an initial LVM volume named `cinder-volumes` as the `kolla-ansible` wrapper will expect an existing volume group with the mentioned name as stated in the `globals.yml` file.

Object storage service

Swift is made up of several subcomponents to manage objects, including accounts, containers, and objects. At the highest level, the main interface to interact with the rest of the object storage ring is the `swift-proxy-server` daemon, which will be running on the cloud controller instance(s).

Important note

Swift is disabled by default in the current `kolla-ansible` infrastructure code repository. It can be enabled by setting `enable_swift` to `yes` in the `/kolla-ansible/etc/kolla/globals.yml` file.

File-sharing service

As we have aimed to provide a managed shared file service in our initial production draft, a few components of the Manila service will be part of the control plane, including `manila-api` and `manila-scheduler`. Like Cinder, the backend storage flavors of the file-sharing service can vary. We will cover more configuration details of the file-sharing service in *Chapter 5*, *OpenStack Storage – Block, Object, and File Shares*.

Important note

The file-sharing service is disabled by default in the current `kolla-ansible` infrastructure code repository. Enabling the Manila service can be done by adjusting the `enable_manila` setting to `yes` in the `/kolla-ansible/etc/kolla/globals.yml` file.

Monitoring

As stated in *Chapter 1*, *Revisiting OpenStack – Design Considerations*, Ceilometer has been changed to act solely as a metric collector of resources running under the OpenStack command. The `ceilometer-api` component is part of the control plane. Considering the way Ceilometer functions, its agent-based architecture involves the existence of data collection agents that collect metric data via REST APIs and poll resources. In addition to `ceilometer-api`, a ceilometer polling agent also runs

in the controller node with a central agent to poll all resources created by various tenants under the OpenStack infrastructure. A notification agent listening to the message bus to consume notification data emitted by agents can optionally run on the cloud controller node.

Important note

The Ceilometer service is disabled by default in the current `kolla-ansible` infrastructure code repository. Set `enable_ceilometer` to `yes` in the `/kolla-ansible/etc/kolla/globals.yml` file. The default playbook configuration of Ceilometer is located under the `/kolla-ansible/ansible/roles/ceilometer` directory.

Alarming

Alarming was originally designed as part of the Ceilometer features and was decoupled to run on its own project with the code name **Aodh**. The alarming components can live within the cloud controller node, which includes additional components for alarm evaluation, a notifier, and a state database.

Important note

The Aodh service is disabled by default in the current `kolla-ansible` infrastructure code repository. Set `enable_aodh` to `yes` in the `/kolla-ansible/etc/kolla/globals.yml` file. The default playbook configuration of Aodh is located under the `/kolla-ansible/ansible/roles/aodh` directory.

Dashboard service

Horizon runs behind an Apache web server instance that serves as a frontend to expose OpenStack services in a dashboard, but not all non-core services are reflected in the dashboard by default. Horizon needs to be configured by enabling directives for certain services and features, such as VPN as a service. Horizon is based on the Python Django framework, which could be overutilized if the number of end users keeps increasing and requests are performed excessively. Scaling the dashboard is something one should expect, especially if most of the supported directives for additional services are enabled and exposed to a wider audience.

Important note

Horizon is counted as a core service in the `kolla-ansible` infrastructure code repository by default. The `enable_openstack_core` stanza configuration is set to `yes` by default, which includes the Horizon service in the `/kolla-ansible/etc/kolla/globals.yml` file. In the same file, you can enable additional Horizon directives to be managed through the dashboard by commenting out the `enable_horizon_service` configuration line where the service refers to the additional OpenStack service. By default, all additional services supported by the dashboard are enabled in the Horizon playbook, as reflected in the `/kolla-ansible/ansible/roles/horizon/default/main.yml` file.

As we have covered most of the OpenStack services that will be part of the control plane, the next section will iterate through the non-OpenStack services counted as shared services.

Shared services – infrastructure services

In *Chapter 1*, *Revisiting OpenStack – Cloud Design Considerations*, we identified shared services as being non-OpenStack services. Shared services, such as queuing messages, databases, and, optionally, the caching layer, are the most critical ones besides other core services.

Message queue

As part of the control plane, RabbitMQ is the most commonly used messaging service in OpenStack. Dedicating separate nodes for the queue is highly recommended but is not a *must-have*. Vast deployment based on multi-regional OpenStack environments is a suitable case to run in a dedicated RabbitMQ cluster due to the meshed requests of different services, including compute, storage, and networks.

The other overlooked aspect of the queuing message is the security requirement. RabbitMQ sits between most OpenStack services as a communication hub. It is important to keep message exchanges secured. RabbitMQ supports TLS, which protects subscribers and clients from tampered-with messages and forces encryption in transit.

> **Important note**
>
> The RabbitMQ TLS option is disabled by default in the `kolla-ansible` infrastructure code repository. It can be enabled by setting `rabbitmq_enable_tls` to `yes` in the `/kolla-ansible/etc/kolla/globals.yml` file. Additional RabbitMQ configurations can be adjusted in its associated playbook, such as message and queue expiration, which are located in the `/kolla-ansible/ansible/roles/rabbitmq/default/main.yml` file.

Database

The other vital piece of the OpenStack puzzle ecosystem is the database holding updates of all service states and user-tracking information. MariaDB is considered the database engine as part of the control plane, made better with the Galera wrapper for clustering in multi-master cluster mode.

> **Important note**
>
> MariaDB clustering relies on HAProxy for database instance switching. More details will be discussed in *Chapter 7*, *Running a Highly Available Cloud – Meeting the SLA*. The default database engine is configured with MariaDB in the `kolla-ansible` infrastructure code repository.

Caching

Caching comes as an optional piece of our control plane. However, it is strongly recommended to begin with caching due to its light footprint in cloud controller utilization. Memcached is becoming a standard to provide ephemeral data between different OpenStack services, such as API interaction information between the identity service and the rest of the services demanding token validation.

> **Important note**
>
> Memcached is enabled in the `kolla-ansible` infrastructure code repository by default. The maximum cached memory size and number of connections can be set in the Memcached playbook located in the `/kolla-ansible/ansible/roles/memcached/default/main.yml` file.

Identifying the control plane of OpenStack is an essential step before starting the deployment of the private cloud. That will help to identify which design layout will fit in your environment to ensure high availability as the highest priority. The next section will iterate through the next design draft scoped around the OpenStack control plane.

Arming the control plane – prepare for failure

The composition of different components around the OpenStack control plane requires more effort to keep them up and running with maximum resiliency. There are a variety of ways to achieve maximum availability for the control plane, but they should be prepared and tested in advance. Within the latest releases of OpenStack, several tools have been adopted to ensure a scalable and highly available control plane. Another approach is simply reusing some patterns that already exist, such as highly available database layouts or messaging bus clustering. We will discuss each of the OpenStack layers' high availability in more depth in *Chapter 7*, *Running a Highly Available Cloud – Meeting the SLA*. It is essential, at this level, to define a strategy to empower the control plane from both scalability and availability perspectives.

Both terms should be considered from the first iteration of the design plan by addressing the following questions:

- How can a control plane respond to a large number of API requests?
- How can hardware failure be faced and resolved in the least amount of time?
- Which design patterns can be used to ensure a stable and resilient control plane?
- How can we handle the growth of infrastructure services such as databases and messaging buses?

The OpenStack ecosystem has been designed to scale massively. Armed with its modular architecture, designing the control plane for resiliency is an open door for different ways of implementation. A summary of possible known design options to prepare for a successful control plane deployment is as follows:

- **Active/active**: All cluster nodes participate in processing requests. This pattern is mainly associated with an *asymmetric clustering* layout whereby an entry point such as a load balancer distributes the requests across active control plane services. An active/active setup increases the performance and scalability of the cloud infrastructure by distributing the load across an army of nodes. Automating the subscriptions or discarding a node can be orchestrated with a load balancer and does not necessarily require a resource manager depending on the load balancer's capabilities.
- **Active/passive**: Unlike the previous mode, active nodes serve requests while standby nodes take over only when one or several active nodes fail. Passive control plane nodes are referred to as *sleepy watchers* and do not require an additional frontend point such as a load balancer, unlike in active/active mode. Such designs are associated mainly with *symmetric clustering* whereby resource manager software evaluates and ensures that the active control plane component is the only one serving the requests at any point in time.

Important note

The `kolla-ansible` infrastructure code repository comes with predefined roles for a highly available control plane defined in the `globals.yml` file with the `'hacluster'` stanza configuration. Most of the adopted clustering software tools within the OpenStack control plane services, including the Antelope release, use a few **HAProxy** instances for load balancing and Keepalived for dynamic cluster node health checks based on the concept of **Virtual IP** (**VIP**) addresses. The HAProxy playbook comes with the latest customizable configuration settings located under the `/kolla-ansible/ansible/roles/loadbalancer/` directory. *Chapter 7*, *Running a Highly Available Cloud – Meeting the SLA*, will cover the HAProxy and Keepalived setup in more detail to deploy a highly available OpenStack control plane.

Designing for control plane failure would depend on the target **Service-Level Agreement** (**SLA**) and available options within your infrastructure. Active/active mode could be a *best-effort* scalable and highly available implementation but would cost extra by dedicating either a hardware or software appliance to assign the load-balancing role of the cluster. The choice of database is another dilemma that most cloud architects and operators pay special attention to due to complexity and performance concerns. For example, an organization's policy dictates the usage of the Oracle database engine. Setting an active-active database setup might not be straightforward to address control plane high-availability requirements because that might require more specific tools and additional integration, unlike using other engines such as MySQL Galera.

Planning for more

Relying on the CI/CD tool as a source of truth is a best practice to keep a consistent control plane services cluster. As we will cover in *Chapter 7*, *Running a Highly Available Cloud – Meeting the SLA*, adding a new cloud controller node should be done transparently without disturbing the running API services. That is another valid reason to develop more integration tests of service consistency during a new control plane node deployment in a separate environment. A monitoring system should feed back the capacity, service availability, and resource usage to ensure a successful production day if all is as expected.

The process of deploying an additional cloud controller node to empower the control plane can be applied the same way when joining additional control plane services. Deploying more OpenStack services will expose more APIs to interact with the other nodes and handle more user or service requests. This can increase the load by hundreds of API requests per minute, and undoubtedly more CPU, memory, networking, and disk will be consumed. Raising performance and scalability concerns for such matters should not limit the growth of your OpenStack cloud services portfolio as there are several ways to address such a challenge. As long as automation is preserved throughout all operations tasks, the deployment of a new OpenStack component that counts as a new control plane service should pass through the CI/CD stages. The new component should pass through the testing and development environments and send feedback with a few metrics such as capacity and usage limits before promoting it to the production environment. A design decision can be made afterward, for example, by running the new component in the same cloud controller cluster or dedicating a separate physical or virtual node to run.

The addition of a new API service as part of the control plane should be reflected across the nodes running the specific service role. That is where the CI/CD system will shine and ensure consistency when dealing with the exponential growth of the infrastructure in terms of nodes and services.

> **Important note**
>
> A few other OpenStack services will be covered and integrated in the next chapters as part of the control plane, such as **Magnum** and **Zun** in *Chapter 4*, *OpenStack Compute – Compute Capacity and Flavors*; LBaaSv2 in *Chapter 6*, *OpenStack Networking – Connectivity and Managed Service Options*; and Watcher in *Chapter 9*, *Benchmarking the Infrastructure – Evaluating Resource Capacity and Optimization*.

Extending the deployment

Once different services that construct the control plane have been identified, it is a good time to update the initial design draft. An ideal way to start deployment with the upcoming production stage in mind is to assign each physical node a set of roles to run different OpenStack services. That brings us to the following configuration:

- **Cloud controller node**: This runs the control plane services including OpenStack APIs, identity, imaging, file sharing, telemetry, and dashboard services. Shared services, including a database with a MySQL Galera cluster, the RabbitMQ messaging service, and Memcached, will be part of the assigned cloud controller node role.
- **Compute node**: This runs KVM as the main hypervisor with Nova compute and networking agent services.
- **Network node**: This runs Neutron agent services, including `layer3`, `lbaas`, `dhcp`, and plugin agents.
- **Storage node**: This runs Cinder volume services interfacing with the Cinder scheduler to provide block storage resources.
- **Deployer node**: This runs the orchestration deployment toolchain, including Ansible, the Kolla builder, and the CI server. The deployer node will optionally host a local private Docker registry.

The updated design draft exposes a multi-node OpenStack environment, as shown in the following diagram:

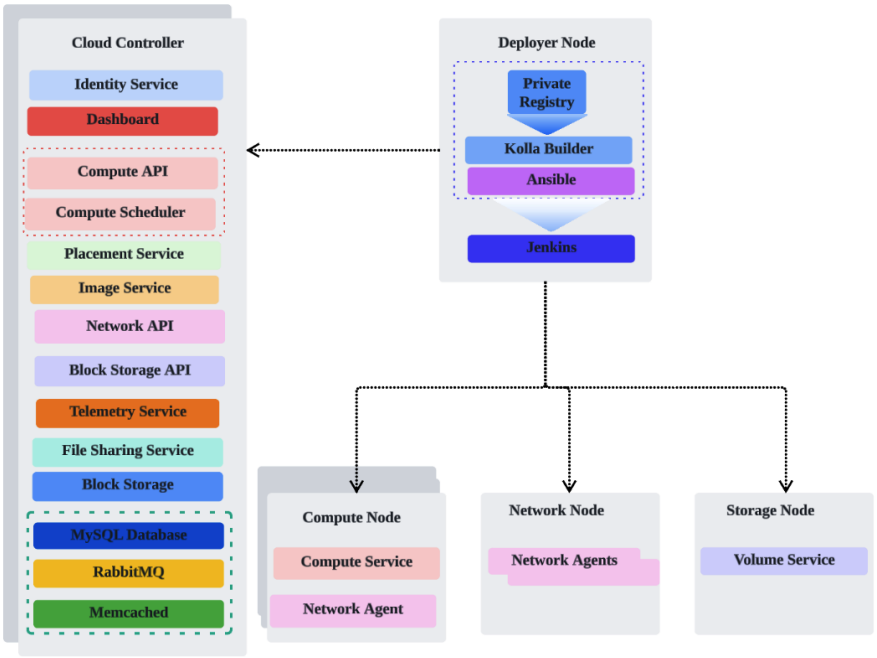


Figure 3.5 – A multi-node OpenStack deployment layout

Important note

Depending on the availability of your hardware, the network and storage roles can be assigned to the cloud controller node. Use the monitoring metrics data to decide in advance the right moment when those services should be installed in separate physical nodes. Splitting storage and network services is recommended when dealing with the exponential growth of the OpenStack infrastructure usage. If you are planning to run the nodes in a virtual environment using tools such as VMware ESXi or VirtualBox, make sure to have the hardware virtualization extensions installed in addition to the virtual networking configuration.

Initializing the network

As depicted in *Chapter 1*, *Revisiting OpenStack – Design Considerations*, a few networks should exist and be prepared in advance. Preparing the physical networking beforehand is essential as part of the extension of the production layout in the next iterations. The proposed standard networks in our initial draft include management, tenant, storage, and external networks connecting different nodes in our initial cluster layout.

The following interfaces will be considered:

- `eth0`: The management interface
- `eth1`: The overlay and tenant interface
- `eth2`: The external interface
- `eth3`: The storage interface

As shown in the following diagram, four distinct types of traffic are considered for the OpenStack networking setup, including management, tenant and overlay, storage, and external. Note that a network node is dedicated to managing various OpenStack networking services that would increase the performance with fewer chances of network saturation at the cloud controller node level:

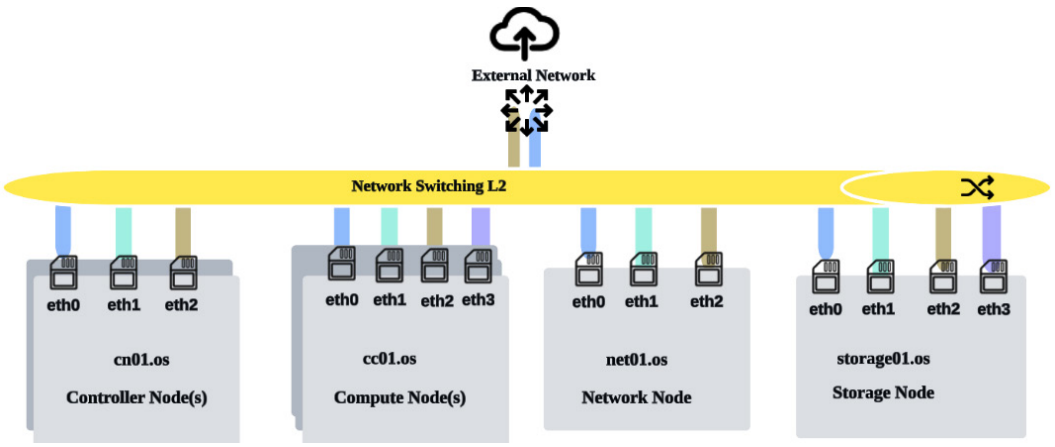


Figure 3.6 – Physical network segmentation layout

At the switching layer, networks are organized logically through the VLANs, where each network is assigned a VLAN ID coupled with the assigned network IP subnetwork address pool. The following table provides an example of initial network mapping given a reserved IP pool of `10.0.0.0/8`:

Network name	IP Pool	VLAN ID
Management	`10.0.0.0/24`	`100`
Overlay and tenant	`10.10.0.0/24`	`200`
External	`10.20.0.0/24`	`300`
Storage	`10.30.0.0/24`	`400`

Table 3.1 – The OpenStack network IP addresses

Important note

When managing OpenStack packages internally, the management interface does not require outbound access to the external network to reach the internet to download packages. The deployer instance can be connected to the management network within the same VLAN to install the required packages directly. External network access can be limited to the deployer instance and tenant network for future instances of external access.

As we are dealing with OpenStack services running in Docker containers, it is essential to take the created virtual network components under the hood into consideration. Once the OpenStack environment is deployed, Kolla will use the Neutron networking model to create a few bridges, ports, and namespaces across each node in the cluster. As we will see in the next configuration section, Kolla maps each network interface to a Neutron physical network denoted as `physnet`. Each container bridge created by Neutron is defined in the `neutron_bridge_name` stanza code configuration to set the default Kolla naming convention stated as `br_ex`. As mentioned earlier, the `kolla-ansible` deployment mode is highly customizable, so we can assign meaningful names for each bridge mapped to each network interface, respectively.

Important note

The assigned network interface will be used by Neutron to access the public network to allow tenants to create routers and assign floating IPs to workloads requiring internet access defined in the Kolla `/etc/kolla/gloabls.yml` configuration file by assigning the `neutron_external_interface` setting to the dedicated interface, `eth2`, on the compute node.

The network bonding layout is highly recommended to be configured in advance for each host. As discussed earlier, the bonding configuration depends on the tools and running software when using Linux as an operating system that varies from one distribution to another. Commonly used bonding configurations are the native Linux bonds and Open vSwitch.

Important note

Bonding configuration is out of the scope of this book. A good source of bonding configuration for Ubuntu using the kernel module can be found here: `https://help.ubuntu.com/community/UbuntuBonding`

Listing the cluster specs

The initial draft of the physical layout in *Chapter 1*, *Revisiting OpenStack – Design Consideration*, can be considered as a starting point to pick up our hardware specs for each host. With slight differences between host roles, the following table summarizes a typical production hardware and software setup for each host:

Hostname	CPU cores	RAM	Disk	Network	Software	Role
`cc01.os`	8	128 GB	250 GB	4 x 10 GB	Ubuntu 22 LTS, OpenSSH, Python 2.7 or greater, NTP client	Cloud controller
`cn01.os`	12	240 GB	500 GB	4 x 10 GB	Ubuntu 22 LTS, OpenSSH, Python 2.7 or greater, NTP client	Compute node
`net01.os`	4	32 GB	250 GB	4 x 10 GB	Ubuntu 22 LTS, OpenSSH, Python2.7 or greater, NTP client	Network node
`storage01.os`	4	32 GB	1 TB	4x10 GB	Ubuntu 22 LTS, OpenSSH, python2.7 or greater, NTP client	Storage Node

Table 3.2 – OpenStack hosts configurations

Note that OpenStack is designed to run on commodity hardware. The table did not mention any precise hardware model that would depend on the set budget and resources to kick off a first setup. As we planned, in the first chapter, to accommodate 200 instances with the *small* flavor during our first deployment, the production setup should be capable of being expanded with more hardware and scale horizontally for each layer of the OpenStack ecosystem. The **undercloud** monitoring tools should regularly gather data on the performance and usage of each node to help in deciding the right sizing and additional resources needed to be plugged in. The first hosts and hardware listings should not welcome any production workload before arming the cluster with a layer of redundancy that will be discussed in detail in *Chapter 7*, *Running a Highly Available Cloud – Meeting the SLA*.

Preparing the infrastructure code

Using the same `kolla-ansible` repository, we can customize the setup of different OpenStack services, including the control plane services, in the same way as highlighted in *Chapter 2*, *Kicking Off the OpenStack Setup – The Right Way (DevSecOps)*.

As a best practice, use a separate branch before merging to master and then start the deployment into the production environment.

Configuring the host groups

The `kolla-ansible` infrastructure code defines the target hosts and associated parameters such as hostnames and network IP addresses in the `/ansible/inventory` directory. The listing of host groups can be customized based on roles associated with the initial draft. The format of an inventory file is straightforward as highlighted in *Chapter 2*, *Kicking Off the OpenStack Setup – The Right Way (DevSecOps)*. The following table defines the different host groups in our next deployment iterations:

OpenStack Kolla – Ansible host groups	**OpenStack node role**
`control`	Runs most of the OpenStack control plane, including core service APIs, database, and messaging queue services
`compute`	Hosts running the `nova-compute` service
`network`	Dedicated to network nodes running Neutron plugins and agents
`haproxy`	Target hosts running HAProxy
`storage`	Runs the Cinder volume service
`monitoring`	Dedicated host(s) for monitoring OpenStack servers running Prometheus and Grafana
`logging`	Runs and processes logs for different OpenStack services
`deployment`	Runs the deployment of the infrastructure code using Ansible and Kolla
`object`	Runs the Swift object storage server's instances

Table 3.3 – OpenStack Ansible host groups

The inventory file is highly customizable and can be adapted to your needs depending on whether you need to accommodate different services in a new role or separate more core services in dedicated hosts. Our next step will consider the deployment of a multi-node setup, including one dedicated host for each cloud controller, compute, storage, and network role, respectively. This deployment should not be counted as production-ready as each role will be deployed in a single node that does not yet meet the high-availability aspect. We will cover high availability and fault tolerance in OpenStack deployment in *Chapter 7*, *Running a Highly Available Cloud – Meeting the SLA*.

Starting with the source of truth, the deployer node where the CI/CD chain lives, create a new branch dedicated to a staging environment:

```
$ git branch staging
```

Create a new file named `multi_packtpub_prod` in the cloned repository, `/ansible/inventory`. A simple way to assign different host groups from a default multi-node configuration file can be found in the same directory, `/ansible/inventory`, named `multinode`.

The configuration for each host group is formatted as follows:

```
[role_name1]
Hostname01
Hostname02
…

[role_name2]
Hostname03
Hostname04
…
```

The inventory file can be customized in different ways by assigning explicitly a single or several OpenStack services to run on specific hosts. That can be accomplished by adding the following configuration stanza:

```
[service_name:children]
Hostname
```

For example, the following configuration instructs `kolla-ansible` to install the Swift service on the cloud control node group assigned as control:

```
[swift:children]
control
```

The previous configuration stanza will install and run all Swift components in the control host group. With a more granular configuration, it is possible to instruct which component will be installed in which host. If none is mentioned, the service will be installed in the control host group. The next example instructs `kolla-ansible` to install other components rather than `swift-proxy-server` in the host group referenced as `object_storage`:

```
[swift-proxy-server:children]
swift
[swift-account-server:children]
object_storage
[swift-container-server:children]
object_storage
[swift-object-server:children]
object_storage
```

For our next iteration, the inventory section pointing to the shared services such as the database caching layer and messaging queue is as follows:

```
[control]
cc01.os.packtpub
[mariadb:children]
control
[rabbitmq:children]
control
[outward-rabbitmq:children]
control
[memcached:children]
control
```

Core control plane services will be running on the cloud controller host group, as follows:

```
…
[keystone:children]
control
[glance:children]
control
[nova:children]
control
[placement:children]
control
[cinder:children]
control
[neutron-server:children]
control
[horizon:children]
control
[swift:children]
control
[manila:children]
control
[ceilometer:children]
control
```

More granular shared and control plane service configurations can be customized per service, as follows:

- For the image service, Glance can be configured as follows:

```
..
[glance-api:children]
glance
```

- All compute components except `nova-compute` can be configured as follows:

```
[nova-api:children]
nova
[nova-conductor:children]
nova
[nova-super-conductor:children]
nova
[nova-novncproxy:children]
nova
[nova-scheduler:children]
nova
[nova-spicehtml5proxy:children]
nova
[nova-compute-ironic:children]
nova
[nova-serialproxy:children]
nova
```

- The Placement service API can be configured as follows:

```
[placement-api:children]
placement
```

- The block storage API and volume scheduler service can be configured as follows:

```
[cinder-api:children]
cinder
[cinder-scheduler:children]
cinder
```

- The object storage proxy component can be configured as follows:

```
[swift-proxy-server:children]
swift
```

- The telemetry core and notification components can be configured as follows:

```
[ceilometer-central:children]
ceilometer
[ceilometer-notification:children]
ceilometer
```

The shared services and control plane sections can be defined in different ways. More services and components would inherit the same configuration once the design of a new project has been drafted to run in the cloud controller nodes. As we are not covering all services in this chapter, we will stick

to our second iteration defined earlier and more control plane services will be added throughout the next chapters of the book. The inventory file will be the source of truth that identifies the latest version of the services running in the environment.

The next class of the inventory section defines the compute target nodes and the respective components they will be running:

```
…
[compute]
cn01.os.packtpub
```

Optionally, as each compute node would require the collection of metrics for further monitoring, `collectd` will run on the compute host group in addition to the `ceilometer-compute` and `ipmi` components:

```
[collectd:children]
compute
[ceilometer-compute:children]
compute
[ceilometer-ipmi:children]
compute
```

As each compute node would interconnect with other nodes to serve tenant networks, a few Neutron agents, as designed in the previous section, will be running on the compute host group:

```
[ovn-controller-compute:children]
compute

[neutron-ovn-agent:children]
compute
```

Important note

OVN will be discussed in detail in *Chapter 5*, *OpenStack Networking – Connectivity and Managed Service Options*, as being the most commonly used networking service in OpenStack within the latest releases, including Antelope.

The OpenStack networking service is designed to run on its own host, except for its API, as part of the cloud control plane. The following code defines the Network node and associated components in the inventory file:

```
[network]
net01.os.packtpub
```

```
[neutron:children]
network
```

All Neutron services and agents except for its API will run on the network node, as follows:

```
[neutron-dhcp-agent:children]
neutron
[neutron-l3-agent:children]
neutron
[neutron-metadata-agent:children]
neutron
[neutron-ovn-metadata-agent:children]
network
[neutron-bgp-dragent:children]
neutron
[neutron-infoblox-ipam-agent:children]
neutron
[neutron-metering-agent:children]
neutron
[ironic-neutron-agent:children]
neutron
[neutron-ovn-agent:children]
network
```

Additionally, the `manila-share` component will run on the network host group that will use the Neutron plugin to provide network access to the file share resources:

```
[manila-share:children]
network
```

The OVN controller was configured to run on the compute node as well as in the network host group, as follows:

```
[ovn-controller:children]
ovn-controller-compute
ovn-controller-network
```

`ovn-controller-network` inherits the association with network host groups:

```
[ovn-controller-network:children]
network
```

Note that the OVN mechanism uses its own database that can run on the cloud controller host group. The OVN database itself supports three different schemas that can be running on the same cloud controller host group, as follows:

```
[ovn-database:children]
control
[ovn-northd:children]
ovn-database
[ovn-nb-db:children]
ovn-database
[ovn-sb-db:children]
ovn-database
```

The next stanza of the inventory is the storage layer in which initially, a single storage node will be defined:

```
[storage]
storage01.os.packtpub
```

The storage node will run the Cinder volume:

```
[cinder-volume:children]
storage
```

The last part of the inventory file includes the deployer host itself, defined as localhost:

```
[deployment]
localhost       ansible_connection=local
```

Other services can be counted as additional services to install if you count running a custom **Zabbix agent**, for example, in all nodes or specific scanning tools reporting to a central hub. A simple way to do this is by creating a new stanza referring to all host groups defined in the inventory. For example, the following code section instructs Ansible to install a few tools, such as `cron`, and services to manage and collect logs using `fluentd` for Kolla containers running our OpenStack services. Optionally, `kolla-ansible` comes with `kolla-toolbox` to quickly run specific command lines in a Docker container once installed:

```
[common:children]
control
network
compute
storage

[cron:children]
common
```

```
[fluentd:children]
common
[kolla-logs:children]
common
[kolla-toolbox:children]
common
```

Important note

In a multi-node setup, make sure that the deployer machine can reach all other hosts to run the Ansible playbooks. It is possible to specify in the inventory file how Ansible should interact with each target host. In the current setup, SSH keys have been generated and pushed to each target host to run Ansible against during the deployment. Bear in mind that hostnames should be propagated in `/etc/hosts` in each node if IP addresses are not used in the inventory file.

Save the file, commit it, and push it to the created branch:

```
$ git add multi_packtpub_stg
$ git commit -m 'Multi Node Inventory - Staging'
$ git push --set-upstream origin staging
```

Once the host mapping and branching configuration are properly done, it is time to dive deep into the `kolla-ansible` code and customize the future OpenStack parameters for different services.

Parameterizing the environment

The next key file is the `globals.yml` configuration file defined in the `/etc/kolla/` directory. This file acts as the central configuration hub for all OpenStack environments. The default file that comes with the community `kolla-ansible` repository defines most of the OpenStack services configuration for each service option. You can copy and comment out the desired configuration line or generate a new one holding the OpenStack configuration settings.

The `globals.yml` file in the current iteration includes the following configurations:

- Use Ubuntu as the Kolla base container distribution:

  ```
  kolla_base_distro: "ubuntu"
  ```

- Install the latest stable OpenStack release:

  ```
  openstack_release: "master"
  ```

- Assign an unused default VIP to one controller interface:

  ```
  kolla_internal_vip_address: "10.0.0.47"
  ```

- Use KVM as the default hypervisor driver:

```
nova_compute_virt_type: "kvm"
```

- Use the Neutron Open vSwitch plugin:

```
neutron_plugin_agent: "openvswitch"
```

- Enable Neutron provider networks:

```
enable_neutron_provider_networks: "yes"
```

- Assign a network interface for API services:

```
network_interface: "eth0"
```

- Assign a network interface for the Neutron external port:

```
neutron_external_interface: "eth1"
```

- Use the local Docker registry as created in *Chapter 2*, *Kicking Off the OpenStack Setup – The Right Way (DevSecOps)*:

```
docker_registry: "docker-registry:4000"
```

- Save the file, commit, and push it to the created branch:

```
$ git add multi_packtpub_stg
$ git commit -m 'Multi Node Global Configuration - Staging'
$ git push staging
```

Important note

Some configuration parameters in the `kolla-ansible` community repository might be updated from one OpenStack release to another, such as new configuration parameters in the `globals.yml` file or the repository structure itself. It is highly encouraged to cast an eye on the repository documentation before deploying a specific release. The book uses the latest stable release. Make sure to check the branch name and its relevant tags for the first pull.

Advanced configuration

The `globals.yml` file does not provide all granular configuration settings by default. For example, the maximum connections of HAProxy and the maximum amount of memory of a Memcached instance are more specific settings that cannot be directly adjusted in the `globals.yml` file.

One way of conducting more advanced configuration in an OpenStack production environment is by going through the Ansible playbooks. Keep in mind that the `globals.yml` file is a high-level representation on how to deploy an OpenStack environment from one single hub file.

There are three different levels of configuring an OpenStack deployment using `kolla-ansible`, in the following order:

- `/etc/kolla/globals.yml`: This is the first hub of generic and custom OpenStack, including shared services settings.
- `/kolla-ansible/ansible/group_vars/all.yml`: This reflects the global and specific settings of the OpenStack services. Once they are edited and running, the settings in the `all.yml` file will be merged with the `globals.yml` file. Settings that are not declared explicitly in the `globals.yml` file will use the default ones in `all.yml`.
- **Ansible playbooks**: Few cases would require the introduction of an advanced configuration in a specific service. Each Ansible playbook should include all available options for a given service or component. Ansible playbooks are located under `/kolla-ansible/ansible/roles`, where each OpenStack component and shared service available within Antelope and later releases are listed. As highlighted in *Chapter 2*, *Kicking Off the OpenStack Setup – The Right Way (DevSecOps)*,in the discussion of how an Ansible playbook is constructed, adjusting a setting for a specific service is straightforward. For example, changing the maximum amount of memory in Memcached can be performed by simply updating the `memcached_max_memory` configuration stanza in the `main.yml` file located under the `/roles/memcached/defaults/` folder.

Customizing the control plane configuration is pretty straightforward by simply looking at the Ansible role and playbook for a given OpenStack core service. `/kolla-ansible/ansible/roles/` includes all sets of OpenStack core services and other projects officially incubated within the latest stable release. Note that there is a direct way of reflecting a playbook setting in the `globals.yml` or `all.yml` file.

Deploying the environment

Dedicating a staging environment to run deployments before running in production is one of the development pillars that is most touted in discussions about code and artifact release stability. There is always a temptation to skip a second environment such as staging when deploying code, especially with resource-hungry environments such as OpenStack. Staging can be expensive, but thinking of the outcome provides a boost of confidence to mimic a production environment before merging that code to the master branch. A common solution to the cost challenge if you are not able to afford an exact copy of the production setup is to use a smaller version of the production environment but keep the same design layout.

In the following snippet, all required steps will be automated through a Jenkins pipeline targeting the `multi_packtpub_stg` branch. To do so, create a new Jenkins file targeting the staging branch:

```
pipeline {

  agent any

  stages {
    stage('Setup Local Environment') {
      steps {
        echo '--RUNNING LOCAL ENVIORNMENT --'
        sh ''' #!/bin/bash
        sudo apt-get update
        sudo apt-get install \
        python3-dev libffi-dev gcc libssl-dev docker.io -y
        sudo apt install python3-pip -y
        sudo apt install python3.10-venv -y
        sudo python3 -m venv local
        source local/bin/activate
        '''
      }
    }

    stage('INSTALLING PIP') {
      steps {
        echo '--INSTALLING PIP --'
        sh ''' #!/bin/bash
        pip install -U pip
        '''
      }
    }
    stage('INSTALLING Ansible') {
      steps {
        echo '--INSTALLING Ansible --'
        sh ''' #!/bin/bash
        pip install 'ansible-core>=2.16,<2.17.99'
        '''
      }
    }

    stage('INSTALLING Kolla Ansible') {
      steps {
        echo '--INSTALLING Kolla Ansible --'
        sh '''#!/bin/bash
```

```
        pip install \
        git+https://git@ci.os/git/openstack_deploy/openstack_deploy
        kolla-ansible install-deps
        '''
      }
    }

    stage('Preparing Infrastructure') {
      steps {
        echo '--Preparing Infrastructure Files Structure --'
        sh ''' #!/bin/bash
        sudo mkdir -p /etc/kolla
        sudo chown $USER:$USER /etc/kolla
        cp -r /usrlocal/share/kolla-ansible/etc/kolla/* /etc/kolla/
        cp -r /usr/local/share/kolla-ansible/ansible/inventory/* \
        /etc/kolla/
        '''
      }
    }

    stage('Secrets Setup') {
      steps {
        echo '--Generating OpenStack Services Secrets --'
        sh '''#!/bin/bash
            kolla-genpwd -p /etc/kolla/passwords.yml
           '''
      }
    }

    stage('Boostrap Servers') {
      steps {
        echo '--Running Ansible Kolla Boostrap Server Script --'
        sh '''#!/bin/bash
             kolla-ansible -i /etc/kolla/multi_packtpub_prod \
             bootstrap-servers
           '''
      }
    }

    stage('Infrastructure Pre-Checks') {
      steps {
        echo '--Running Ansible Kolla Prechecks Script --'
        sh '''#!/bin/bash
```

```
            kolla-ansible -i /etc/kolla/multi_packtpub_prod prechecks
            '''
        }
      }

      stage('Deploy Infrastructure') {
        steps {
          echo '--Running Ansible Kolla Prechecks Script --'
          sh '''#!/bin/bash
          kolla-ansible -i /etc/kolla/multi_packtpub_prod deploy
          '''
        }
      }
    }
}
```

Commit and push the created file to the source code repository with the new branch:

```
$ git add Jenkinsfile
$ git commit -m 'Jenkins Staging Environment Pipeline'
$ git push staging
```

As demonstrated in *Chapter 2*, *Kicking Off the OpenStack Setup – The Right Way (DevSecOps)*, create a new job by selecting **New Item** from the top-left menu in the Jenkins user interface. Select **Pipeline** as the type of our job with a desired name for running the staging pipeline. Follow the same steps for the staging pipeline by specifying the path of the Jenkins file from the repository. Different build stages are visible as coded in the pipeline, as shown in the following figure:

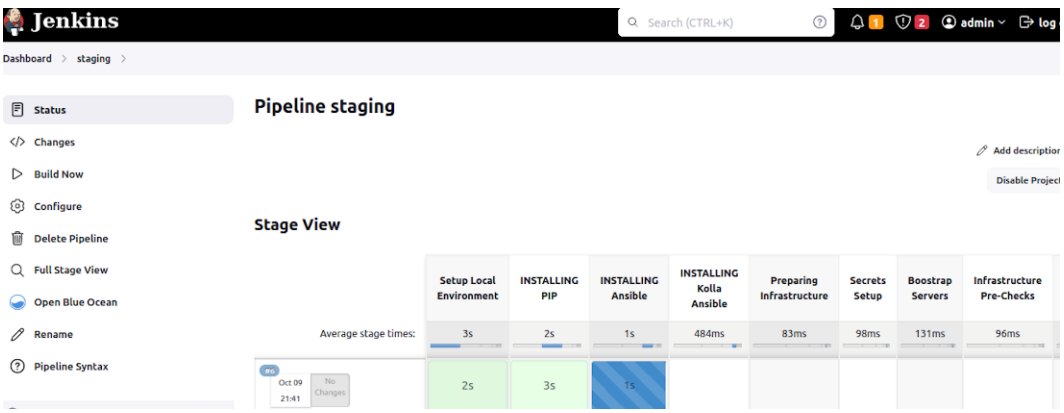


Figure 3.7 – An OpenStack staging environment deployment

The first build run can take longer to install the required packages and dependencies on the Jenkins server. Automating the deployment pipeline is very handy as orchestrating all steps from one single pipeline file reduces the time to run specific steps manually in case an error occurs in one of the pipeline stages. A Jenkins build can be configured to pull source code on a regular basis (`cron`, for example) or trigger a build when new code is pushed and merged to the development branch. Note that there are multiple ways to achieve more robust deployment pipelines by integrating more unit tests. We have, so far, built an initial deployment pipeline. From it, installing additional services or making changes can be performed from one single place and with more control. Throughout the next chapters, we will stick to the same deployment approach armed with best practices by keeping deployments in small batches, continuous, and test-driven.

Summary

In this chapter, we went through a cornerstone of OpenStack deployment: the control plane. Identifying each component role within the control plane layer supports the preparation of a highly available and scalable orchestration layer that will we cover in more detail in *Chapter 7*, *Running a Highly Available Cloud– Meeting the SLA*. Some of the core services, such as Keystone, Glance, and Placement, were covered in more detail to highlight different options that your next production iteration can support through identity consolidation or an imaging backend, for example. Note that the control plane is not limited only to the deployed services we have explored so far in this chapter. The OpenStack ecosystem is still growing with additional services; hence, more APIs can be part of the control plane. From an architectural perspective, the Designate API can be part of the control plane and run in the same cloud controller, whereas other management components can run on a dedicated host to enhance DNS request performance in a large-scale environment.

Most importantly, regardless of which option you choose, the control plane must be prepared for failure and proven to be scalable, as we learned in this chapter. Adding or removing a service unit within the control plane cluster should not be a disruptive operation. That was a valid reason to adopt a container-based deployment approach for our OpenStack production environment. The chapter extended the first deployment using `kolla-ansible` by going into more depth on a variety of control plane playbooks with additional possible advanced settings. Although the chapter did not cover the high-availability aspect of the control plane in detail, the design draft should be capable of enforcing the control plane with fault tolerance before exposing the cloud environment for consumption.

The next chapter will extend our deployment by going through the compute layer. As we have separated our control plane and designed only the compute API and scheduler to be part of the cloud controller, the next step is to dive into the OpenStack compute service and uncover various options for deployment as well as features, including containerization options, within the latest OpenStack releases.

4

OpenStack Compute – Compute Capacity and Flavors

"Magic is believing in yourself. If you can do that, you can make anything happen."

– Johann Wolfgang von Goethe

As a private cloud provider, typical OpenStack IaaS services are the building blocks of its offering to end users. At the heart of the IaaS offerings is the compute service that runs and manages the workload life cycle in different facets. The OpenStack Foundation has committed to enlarging the scope of the OpenStack ecosystem and exposing new services by tailoring the Nova service in different ways. Additional new trending projects such as managed elastic data processing (code-named **Sahara**), managed databases (code-named **Trove**), and a container application catalog (code-named **Murano**) are examples of services that are offered on top of the Nova service.

As a cloud operator, having a solid understanding of the compute farm capabilities and the different segregation methods within the OpenStack environment is a foundational requirement to run a successful private cloud journey with the desired service offerings. At the time of writing this edition, the Nova service had some updates, in addition to the emergence of more pertinent features and services around it. To ensure a safe compute design update based on the draft elaborated on in the first chapter, it will be necessary to run through the new compute service that came out with the latest OpenStack releases starting from the Antelope release. In this chapter, we will go through the following topics:

- Revisiting the Nova compute service from an architecture perspective
- Checking out the latest hypervisors supported by the compute service
- Exploring the concept of compute segregation for massive OpenStack compute deployment
- Demystifying the art of filtering to construct advanced ways of performing compute resource allocations and hosting workloads based on an array of strategies
- Discussing the **Container Orchestration Engines** (**COEs**) and their integration with Magnum

- Learning how to manage and operate containers in the most flexible way with the new, trending Zun OpenStack container service
- Configuring and deploying new compute nodes in a running OpenStack environment using Kolla-Ansible infrastructure as a code paradigm

The compute service components

The compute service is composed of multiple components that take care of incoming compute requests and then launch and manage instances, as depicted in the following diagram showing the Nova architecture:

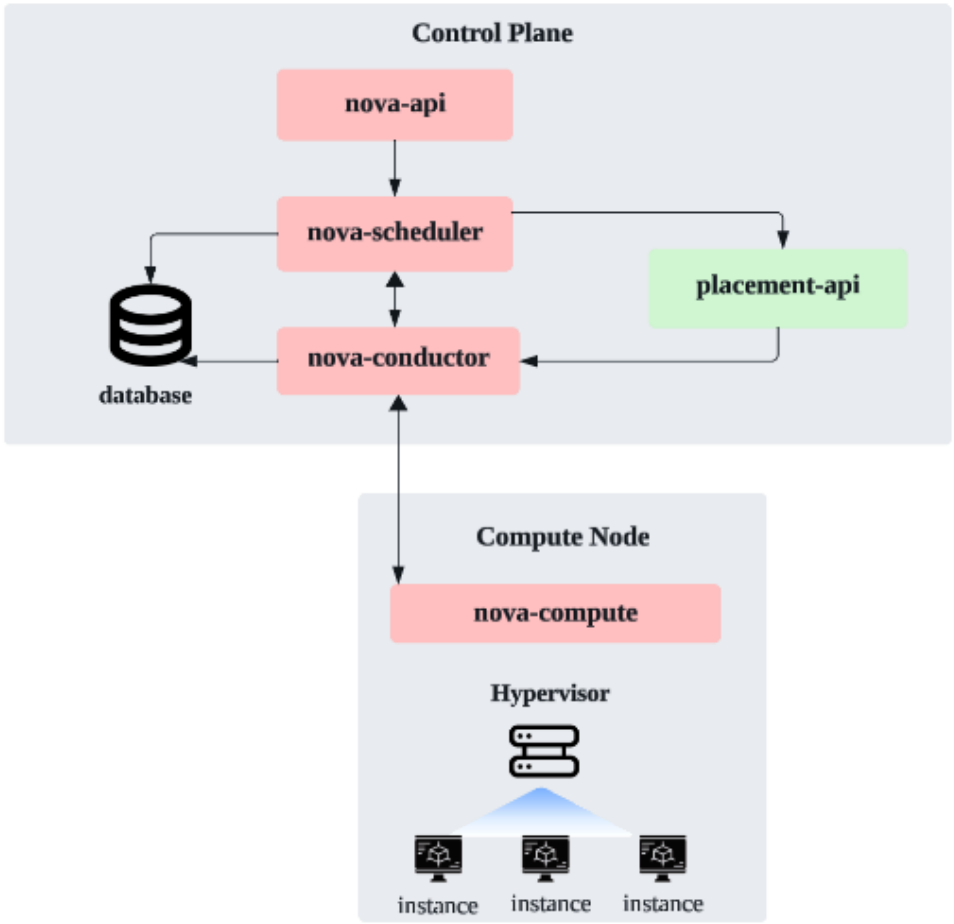


Figure 4.1 – The Nova component architecture

Let's explore each compute component in the following section.

API

The `nova-api` component deals with different (HTTP-based) requests via the queuing message service. Keep in mind that the API is the first interface that accepts compute requests before forwarding them to the next Nova component and establishing a complete workflow.

Scheduler

The next station in the Nova workflow abstraction layer is the `nova-scheduler` component. It is a major piece of the workflow that decides in which compute node each incoming instance creation request will run.

The scheduler component uses a variety of filters and preferences that can be customized to match different needs and reasoning for the availability of the existing resources. **Filtering** and **weighting** are the main mechanisms used by the Nova scheduler in order to customize the allocation of underlying physical resources. This chapter will break down the newly added filters and weighting references in the Antelope and later releases.

Conductor

Once a request is dealt with via the `nova-scheduler` process, the `nova-conductor` component will make sure that the reservation made in the physical host is saved in the shared database. The `nova-scheduler` process will be required to check the database beforehand and verify the available capacity before making a scheduling decision. The conductor component is a major addition to the Nova workflow compared to the earlier releases of the Nova project in the OpenStack ecosystem. As a brief reminder from the previous edition, the conductor comes with a security improvement, preventing the compute nodes from having direct access to the database. In earlier versions, the compute nodes would update their status directly in the database shared between them. That would reduce the blast radius of a possible security gap on the database level if one of the physical machines was compromised. Nowadays, the conductor takes on that responsibility on its own. Another great addition of the conductor component is dedicated operations dealing with the compute nodes, such as resizing, building, and migration.

By having those capabilities in `nova-conductor`, the compute nodes will act as simple worker nodes, leaving coordination and any database operations to the conductor process.

Compute

Coming to the end of the Nova workflow, the `nova-compute` component will take the rest of the requests by dealing with the hypervisor. The addition of a variety of features for different supported hypervisors in the OpenStack compute core makes almost every new release more exciting than the previous one. Libvirt KVM, VMware vSphere, XenServer, XenAPI, Microsoft HyperV, **Linux Containers** (**LXC**), Quick Emulator, and a few others are supported by the Nova service. The most exciting part

is the continuous development and integration of each hypervisor feature by the vendors who own the hypervisor project. This chapter will illustrate more details of those hypervisors and highlight a few common architectural compute layouts.

Console access

Access to virtual machines through OpenStack Horizon (dashboard) or via the **Command-Line Interface** (**CLI**) is well developed by the OpenStack community in different iterations. Console access is not a required component to run a fully working Nova service but is an additional aspect to interact with provisioned guests. Until the *Liberty* release, `nova-console`, `nova-novncproxy`, and `nova-consoleauth` were mainly counted as the most stable console types. Now, with the Antelope and later releases, more console types have joined the list, such as the SPICE console and serial console as an alternative graphical console extension. Microsoft has also contributed to the console service with **Remote Desktop Protocol** (**RDP**). However, the latter comes with a limitation as it supports only Hyper-V and requires integration with the console proxy in OpenStack, which has not been developed. Hence, a third-party tool should be available along with the console configuration, such as the FreeRDP-WebConnect application, which can be found on GitHub. Like Microsoft RDP, VMware has also contributed with another console type, **Mouse, Keyboard, Mouse** (**MKM**). This is bound to a virtual machine's access console based on Vmware's vSphere Hypervisor, but independently of an existing console proxy service in OpenStack. With this growing list in the latest releases, one console-supported service has been deprecated since the Train release, which is `nova-consoleauth`.

The associated Nova API services, including `nova-scheduler`, `nova-conductor`, and `nova-api`, run as part of the controller nodes, and the `nova-compute` service runs on each compute node. Running the Nova compute service is based on the hardware platform and the selected hypervisor, which will be covered in the next section.

Reasoning for the hypervisors

By navigating through the different OpenStack releases, Nova supports a range of **Virtual Machine Monitors** (**VMMs**), also known as hypervisors. Hypervisor software allows access to the physical machine hardware resources and provides different functions to create and manage virtual machines within a compute node. At the time of writing this book, the latest OpenStack releases including the Dalmatian release support **Kernel Virtual Machine** (**KVM**), **LXC**, **QEMU**, **VMware vSphere** (5.1.0 and later versions), **zVM**, **Ironic** (native OpenStack bare metal), **Hyper-V,** and **Virtuozzo** (7.0.0 and later versions). The different features and virtual machine management capabilities differ from one hypervisor to another. A full list of supported features per hypervisor can be found in the official hypervisor support matrix, which is available at `https://docs.openstack.org/nova/latest/user/support-matrix.html`. Make sure to parse each of the provided features based on your requirements and needs. KVM is the most used hypervisor in the OpenStack `nova-compute` setup due to its early adoption and having the largest set of supported features.

Important note

Other famous hypervisors such as **Xen**, **XCP**, **UML**, and the **Docker** driver are not listed in the latest OpenStack releases, including Antelope and Bobcat (released in October 2023). The Docker driver has been supported since the Havana release and moved to its own project, code-named the **Zun** project, which will be covered later in this chapter.

The Nova compute service uses `Virt Driver` to interact with the `Libvirt` library via API calls and manages virtual machines through the hypervisor, as illustrated in the following diagram:

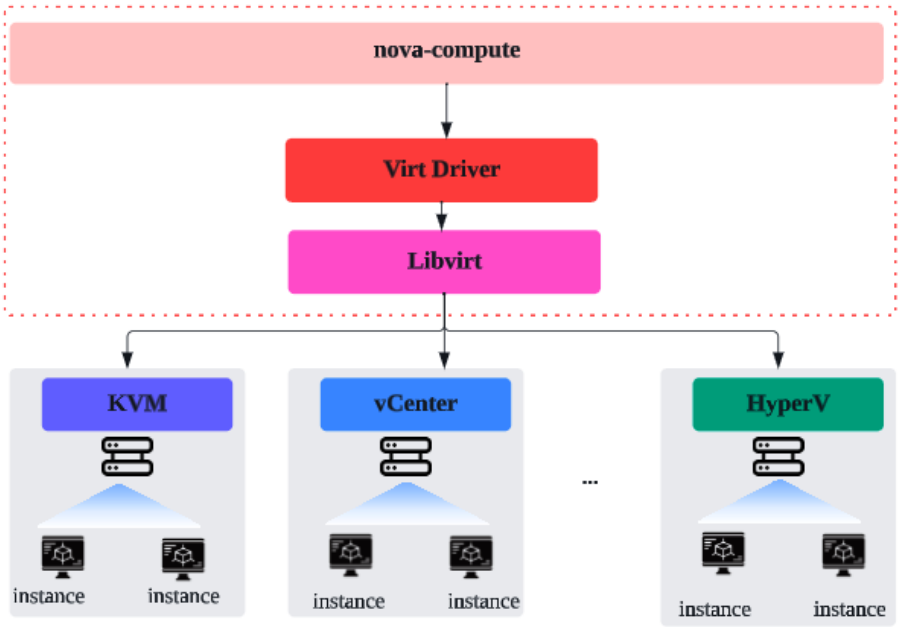


Figure 4.2 – The Nova compute-supported hypervisors

The configuration of the hypervisor backend in Nova can be performed by adjusting the `compute_driver` option in the `/etc/nova/nova.conf` file.

The KVM implementation requires `nova-compute` to be deployed on a Linux server with KVM virtualization modules installed. For each compute node, make sure the KVM modules are properly loaded by checking the following command line:

```
# lsmod | grep kvm
kvm_intel or kvm_amd
```

Optionally, and depending on the supported KVM module (Intel or AMD) in your nodes, add the listed modules in the `/etc/modules` files to persist your changes at reboot.

The next section considers KVM as the main adopted hypervisor of our production compute nodes.

Compute segregation

OpenStack is designed to scale massively to respond to the growth of the infrastructure size and user demand. As the compute service is the workhorse of the cloud infrastructure, designing for compute layer scalability and business continuity is a must. Defining a compute segregation strategy will help you understand your private cloud infrastructure limits and forecast user compute demands – that's what will be explored in the following section.

Infrastructure segregation

Depending on the available compute farm resources and inventory specifications, as well as the hypervisors to be used, OpenStack exposes a variety of compute segregation mechanisms that can be leveraged, including **Availability Zones**, **Regions**, **Cells**, and **Host Aggregates**. Each of the previous concepts will be detailed in the following section.

Regions

A region in the OpenStack glossary presents a fully independent OpenStack deployment. From operational and management perspectives, each region can expose its own OpenStack API services, including compute resources. If you're planning a Nova multi-regional setup, a Nova API endpoint should be created in each region and added to the identity services catalog. Users can spawn instances and spread workloads between available regions through the available compute API. Running workloads in a multi-region configuration setup offers a multitude of advantages for end users, but also presents challenges for operators, such as maintaining a consistent setup of the OpenStack control plane and shared service layers. This is mainly due to the risk of having a **split brain** between compute resources running in more than one region. Hosting a user workload in a fleet of compute nodes running across multiple regions would require certain ways to provide consistency of shared resources such as virtual machine images, databases, or file shares. Another important aspect of multi-regional consistency is the way to handle authorization to request compute resources. One of the common design approaches is to consolidate all authentication calls through Keystone via a federated **Identity Provider** (**IdP**), as shown in the following diagram:

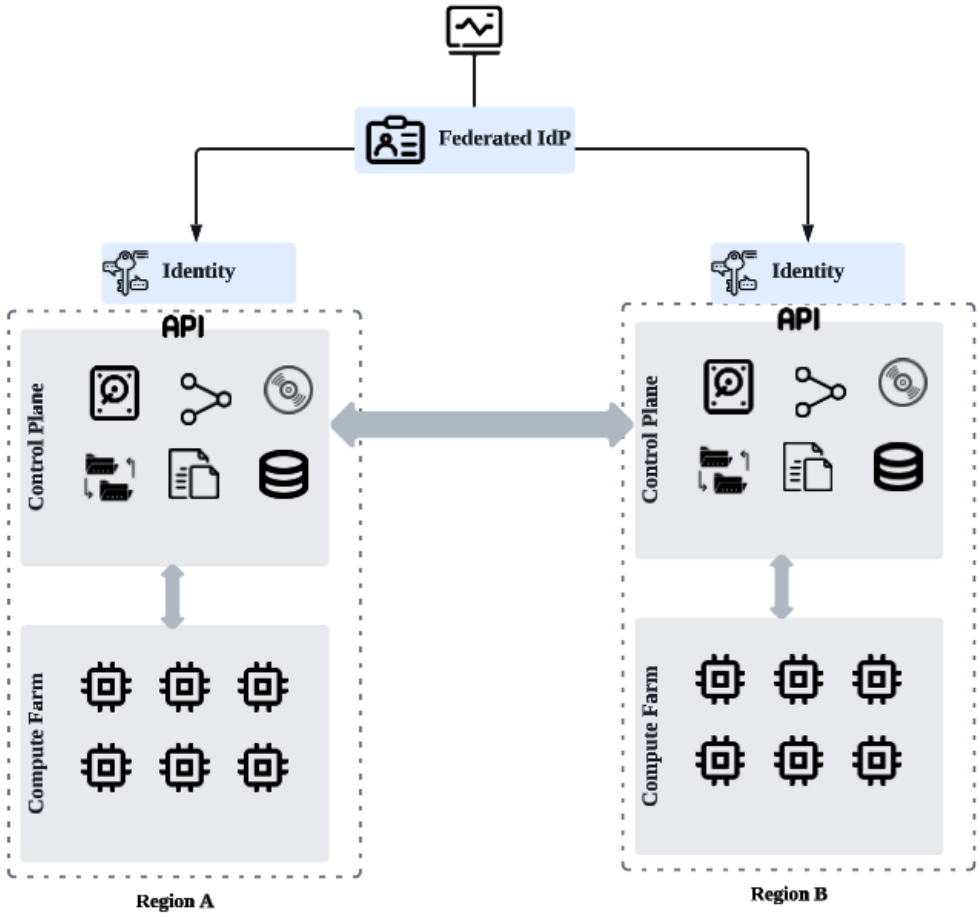


Figure 4.3 – OpenStack multi-region deployment

Each region can host one or more discrete compute structures defined as *Availability Zones*, which will be covered in the next section.

Availability Zones

A region can be logically composed of one or multiple Availability Zones. Unlike the regional setup, there is no need to deploy a full, standalone OpenStack deployment and provide consistency across the control plane services. From a physical implementation standpoint, the compute nodes will be split into trees of logical groups called **fault domains**. Each fault domain will be part of a **Power Distribution Unit** (**PDU**) within a given rack and different hypervisors will run across different racks. This way, losing connectivity to the **Top-of-Rack** (**ToR**) switch or upon a PDU failure won't fully bring down a user workload running in a multi-availability zone configuration, as shown in the following diagram:

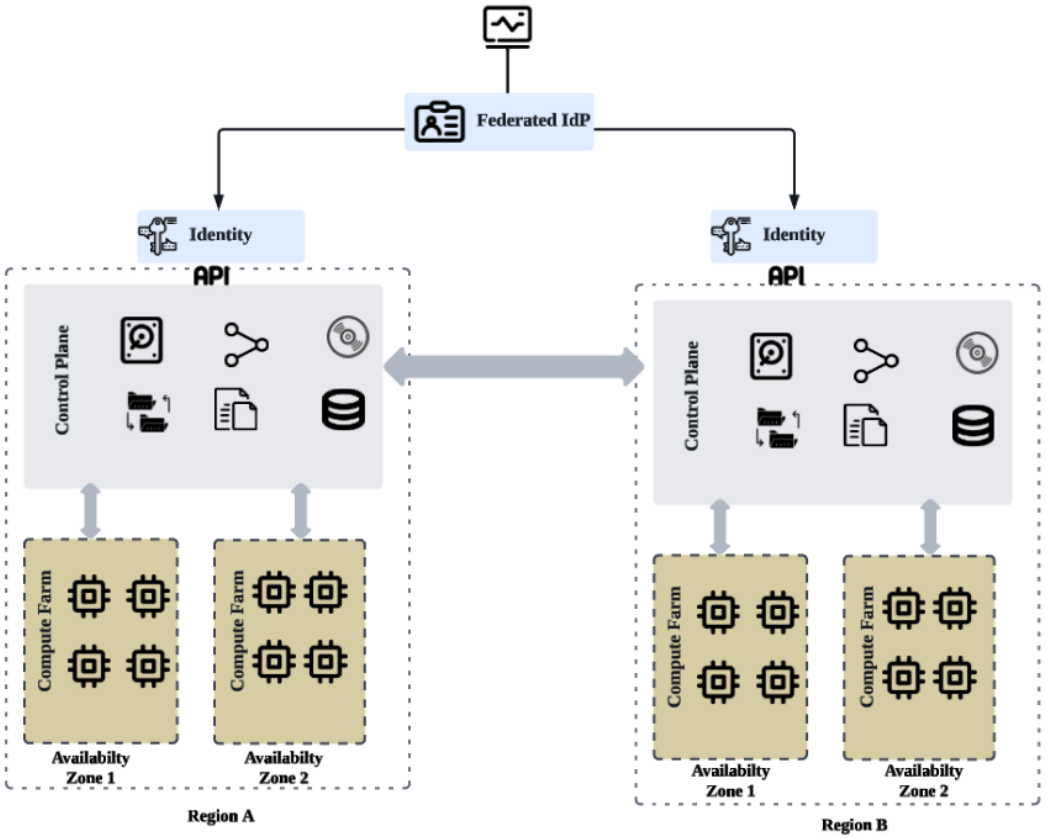


Figure 4.4 – Extended OpenStack multi-regional and zonal deployment

The Nova availability zone option empowers user workloads with high availability that can be considered a premium option for critical workloads that require the highest level of SLA. A compute node can belong to only one availability zone as it is bound to the rack's physical design, such as rack ID, location, and PDU.

> **Important note**
>
> The logical abstraction of Availability Zones covers the network and block storage services besides the compute service.

Users can select which Availability Zone to launch instances in. For this reason, keeping an eye on the compute resource utilization for each Availability Zone is essential. That will ensure that each Availability Zone will be able to accommodate more compute allocation within the available physical resources when needed, as well as ensure that users can spawn workloads across multiple compute nodes in different zones.

Host aggregates

Unlike the Availability Zones concept, the host aggregates strategy groups compute nodes based on more granular hardware profile categories defined as metadata. This type of grouping can be based on the definition of features such as hypervisor, storage type, CPU family, network speed, and so on. Cloud operators should create different host aggregates in advance by attaching a set of metadata to the selected compute nodes. This lets users spawn instances by selecting the desired host aggregate that responds to the workload requirements. Metadata can be created based on plenty of hardware specifications, depending on what the private cloud operator has purchased. For example, aggregation can be configured for an optimized disk host group, based on SSD, GPU architecture, high-performance networking capabilities, or even by aggregate compute nodes for a single tenant. A compute node can be part of one or many host aggregates. As a best practice, it is important to list all the different hardware capabilities once the compute nodes are in place and sort them into trees by grouping them based on the expected workload that they will host. For example, if HPC workloads are included in your cloud business case offering for end users, you might consider creating a host aggregate in advance based on GPU capabilities and exposing it to end users. The following diagram shows a use case for providing four host aggregates, where some compute nodes can be part of more than one host aggregate:

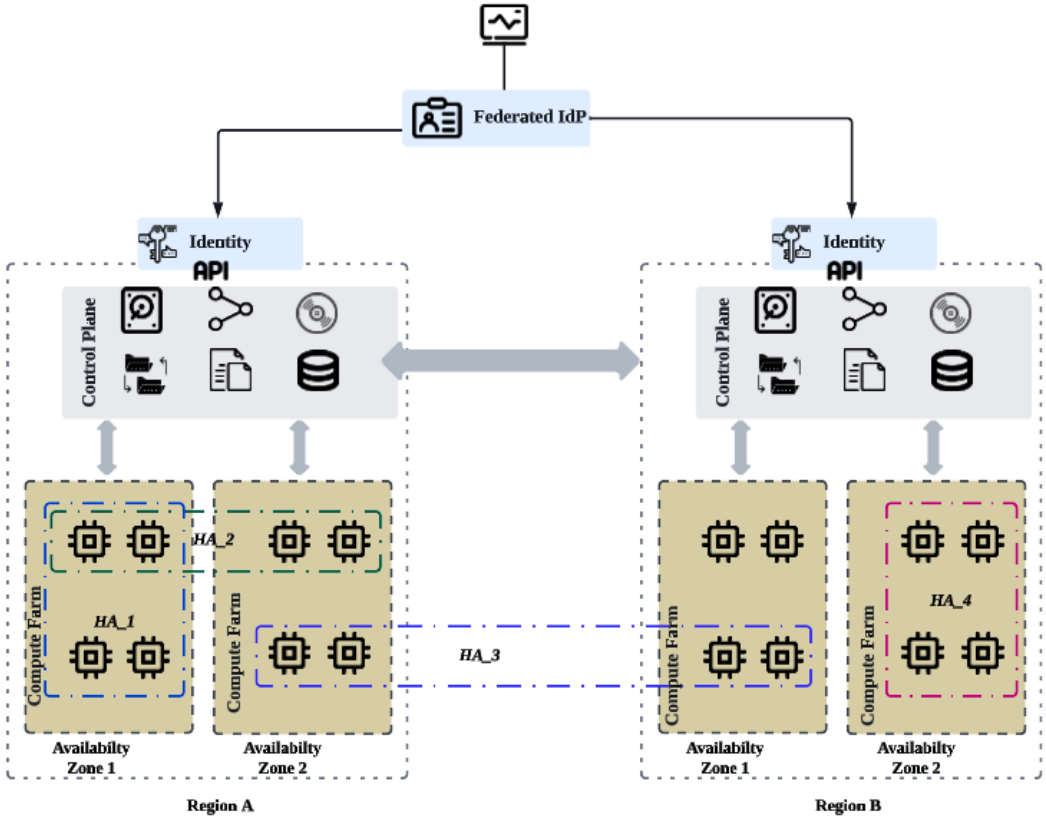


Figure 4.5 – Host aggregation in the deployment of multiple Availability Zones

The previous multi-region segregation compute layout involves four different host aggregates that can be spread across different Availability Zones as follows:

- **Host aggregate 1** (**HA_1**): This involves grouping a set of compute nodes in the same region and Availability Zone to accommodate workloads that require low network-latency performance, such as HPC applications within a large network bandwidth and GPU architecture. Workloads in this host aggregate do not have high availability as a major exigency.
- **Host aggregate 2** (**HA_2**): This aggregate involves grouping a set of compute nodes spread through two Availability Zones and within the same region to accommodate workloads that require large distributed workloads, such as big data processing running Hadoop or Cassandra. The host aggregate metadata includes large network bandwidth, optimized disks based on SSD, and memory capacity.
- **Host aggregate 3** (**HA_3**): This involves grouping a set of compute nodes spread through three Availability Zones and across two regions to accommodate critical workloads that require the highest level of availability and scalability, such as web applications and webshops. The metadata describes moderate network bandwidth, disk, memory, and CPU standard capabilities.
- **Host aggregate 4** (**HA_4**): This involves grouping a set of compute nodes within the same region and Availability Zone to accommodate workloads that require enhanced networking capability within a VSphere environment. The host aggregate metadata includes VMware as a configured hypervisor to extend an existing VCenter environment and spread the application load between VMware hosts (running the `nova-compute` service).

We will demonstrate later in this chapter how the combination of the host aggregation approach, Placement service, and Nova scheduling can expose a gazillion ways to host different types of workloads in the most efficient way to cloud users.

Cells

The concept of cells in OpenStack was introduced mainly to address the performance bottleneck of large-scale Nova deployment within a single OpenStack region. A large number of compute nodes would increase the load on the main control plane services, including the messaging queue and database, resulting in performance degradation that would limit the expansion of the entire system. The mitigation of performance degradation for large-scale deployment for Nova compute resources can be addressed by using the cell approach: compute nodes can be grouped within a logical boundary, where each group runs its own database and messaging queue service. The cell architecture has been developed through different stages, resulting in two versions: **CellV1** and **CellV2**. CellV1 was the first version and was not widely implemented due to a number of caveats. Since the OpenStack Ocata release, CellV2 has been introduced, with enhancements to the former version, and has become the official version within the **Pike** release. Let's discuss the major differences between the versions next.

CellV1

The logical grouping of compute nodes forming each cell is arranged in a tree structure where the top-level cell called the **root** runs the `nova-api` service. The second level of cells, which run the Nova compute service, are referred to as child cells, as shown in the following diagram:

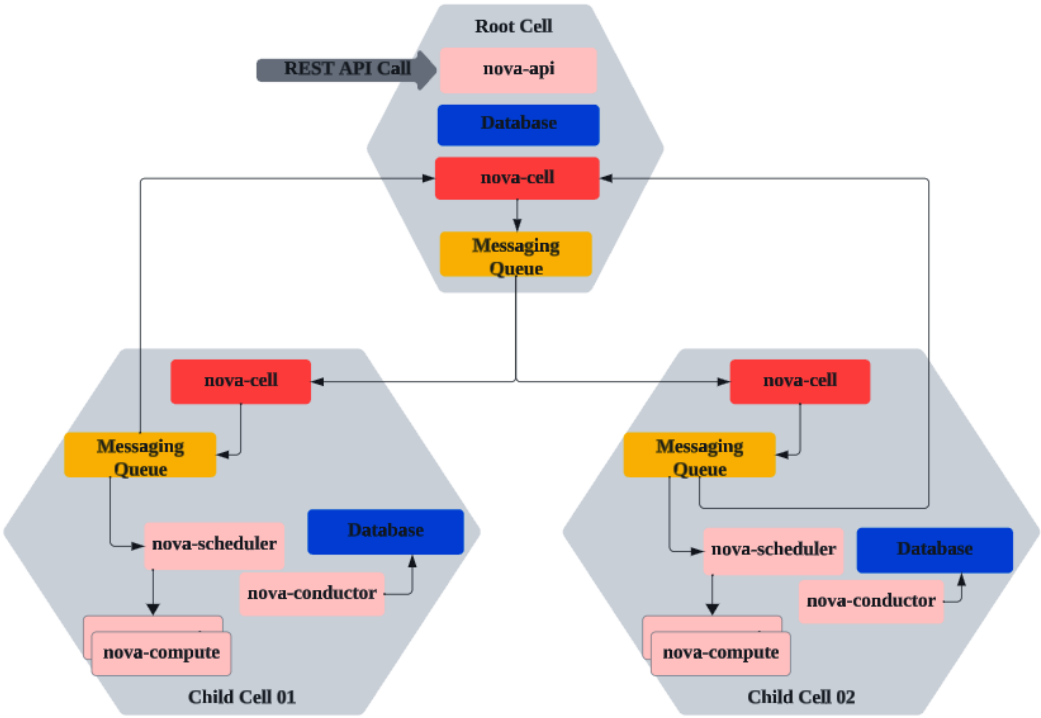


Figure 4.6 – OpenStack compute CellV1 deployment

The interaction between the cells running the Nova API and running the Nova compute goes through a dedicated messaging queue via RPC calls handled by a separate Nova service called `nova-cell` (a major enhancement of CellV1 introduced in the Juno release). Each cell runs its own messaging queue for interactive synchronization and is used by `nova-cell` for selecting in which cell an instance will be spawned. This exposes a higher level of instance scheduling but within the cell's level. Once a cell is selected, the `nova-scheduler` service within that cell will process the rest of the request workflow to select a compute node.

CellV2

The initial cell version went through experimental phases and was not successfully adopted in most large compute deployments. This was mainly due to the multi-layer scheduling to spawn instances from the cell selection down to the compute scheduler service within the selected cell. *Figure 4.6*

illustrates a simple tree composed of only root and child cells, but other, wider deployments could adopt more child cells, called **grandchild** cells. The other major shortcoming of the first version is the amount of data (instance and compute node information) to be mapped and synchronized between cells, resulting in heavy data replication processes between the cells. The new version, referred to as CellV2, has been developed and adopted officially from the Ocata release by addressing the caveats of the prior version. Major changes have been introduced, such as the removal of the `nova-cell` component and the overall cell architecture, which is no longer a tree-like structure similar to CellV1. This is illustrated in the following diagram:

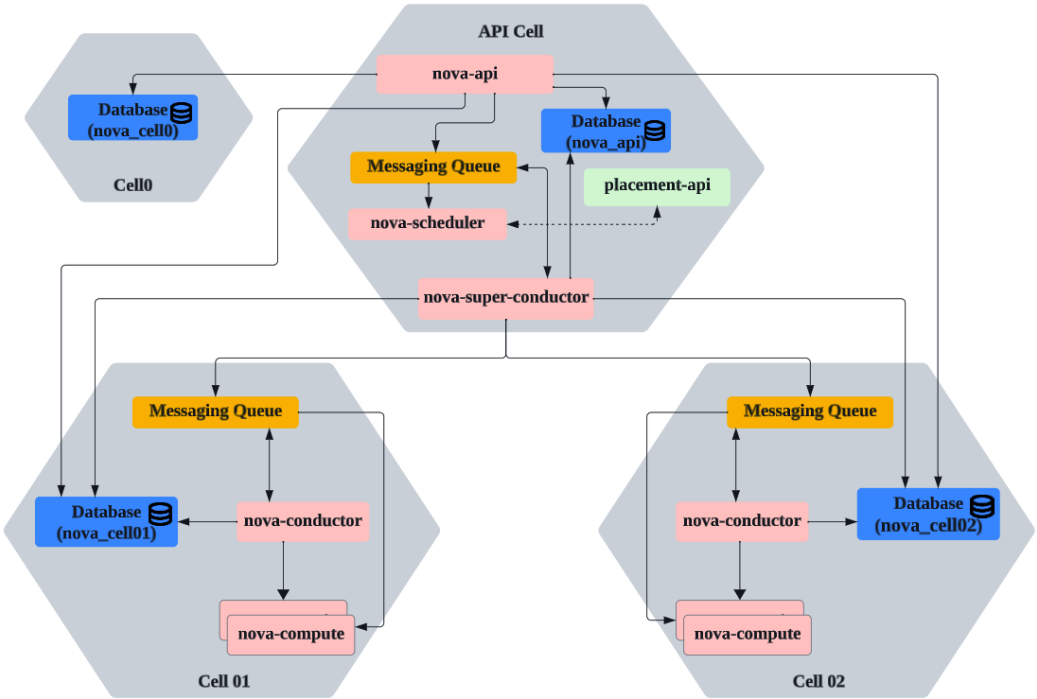


Figure 4.7 – OpenStack compute CellV2 deployment

Sharding hypervisor nodes through the second version of cells in Nova involves major changes in the scheduling workflow within the multi-cell deployment, as illustrated in the previous diagram. A cell API runs the `nova-api` service, `nova-scheduler`, and more recently, the Placement service (the Placement service was not introduced in the OpenStack ecosystem in the Cellv1 development) to uniformly schedule resources across all defined cells. The cell API exposes a separate database called `nova_api`, which contains metadata of all global resources, such as instance flavors, quotas, keypairs, and so on. In multi-cell deployment, the conductor piece in the API cell is called `nova-super-conductor`, which deals with all database insulation. Additionally, from the API level, a special cell called `Cell0` does not run any of the services but holds only the `nova_cell0` database containing information about failed instances to start due to a scheduler failure.

Each cell grouping the compute nodes manages its own resources and they have no affiliation with each other. Each cell has its own database storing information about instances and uses its own dedicated messaging queue to coordinate between the local cell's `nova-conductor` and `nova-compute` services. The new CellV2 method simply involves the following workflow steps to spawn a new instance in a multi-cell deployment:

1. `nova-api` (in the API cell) receives the API REST call and forwards it to the `nova-scheduler` service (in the API cell).
2. `nova-scheduler` contacts the Placement service, applies its filters, and determines which compute node will be assigned.
3. `nova-api` stores the mapping instance information in its `nova_api` database (the `instance_mappings` table).
4. `nova-api` stores the selected compute node record in the database target cell (for example, `nova_cell01` if the selected compute node resides in **Cell01**).
5. `nova-api` sends an RPC call to `nova-super-conductor` to build the instance.
6. `nova-super-conductor` initiates an RPC call to `nova-conductor` in the target cell to spawn the instance via the compute service.

The CellV2 architecture has shifted drastically from the initial CellV1 architecture with the removal of an additional scheduling layer and the division of the data into multiple databases instead of carrying a replication process for each request. This way, expanding a cell perimeter to welcome more compute hosts can be easily performed without the need to go through the whole synchronization process per cell, as we have seen in CellV1. Moreover, the new cell architecture is simple as all cells are equal, making the whole multi-cell deployment a flat-like structure (only two layers), unlike its previous version, which was based on the tree structure. Once the logical layout of cells is designed and ready for deployment, operators will just need to choose which hypervisor nodes will be associated with which cell. Unlike host aggregation, regions, and Availability Zones, users are not aware of the concept of a cell, as the cell strategy is mainly for cloud operators to expand large compute environments with the best management and performance enhancements.

Workload segregation

The latest features of the Nova service provide different strategies to unlock a fully scalable OpenStack compute infrastructure by means of regions, Availability Zones, host aggregates, and cell concepts. To have a full picture of a large deployment, adopting one or more of those approaches would require the definition of an orchestrator element to dispatch instances in a particular set of hypervisor hosts. At the heart of the launch process of instances is the `nova-scheduler` service, which determines on which host an instance will be placed. As highlighted in *Chapters 1* and *3*, the latest OpenStack releases, come with the Placement service, which was introduced in the Newton release. The Placement

service works in tandem with the Nova scheduler for a more advanced and granular host pre-filtering selection process. To make the most of the segregation approaches defined in the previous section, it is essential to skim the surface of different compute scheduling methods that are mindful of the overall compute segregation strategy in your OpenStack deployment.

Pre-filtering

The main motivation for having a dedicated Placement service that runs alongside the Nova scheduler is to resolve the dilemma of a large compute deployment. Prior to the introduction of the Placement service, the Nova scheduler had to loop through all the compute farms, including the enabled filters, in an entire large deployment, resulting in inefficiency and performance issues. Additionally, the sum of resources reported by the compute nodes lacked accuracy due to the basic functions of Nova in calculating, in a detailed fashion, the exact number of resources and inventory usage.

With the advent of the Placement service, the logic of resource reporting has been changed by adding a `resource_tracker` component in a compute node that reports its inventory and available resources to the Placement service, which periodically stores a full synchronization of reported resources. As detailed in *Chapter 3*, *OpenStack Control Plane – Shared Services*, the Placement service comes with the definition of **resource providers** and **traits** to apply prefilters to its compute node list once contacted by `nova-scheduler`. Under the hood, `nova-scheduler` calls the `placement-api` service via a `GET API` request by passing different scheduling parameters such as `VCPU`, `DISK_GB`, `MEMORY_MB`, and `VGPU`. These will be translated into **trait** requests at the `placement-api` level.

Resource provider allocation

The Placement service, as reviewed in *Chapters 1* and *3*, is used by `nova-scheduler` to list a set of pre-filtered compute nodes based on specific attributes before evaluating their candidacy, as shown in the following diagram:

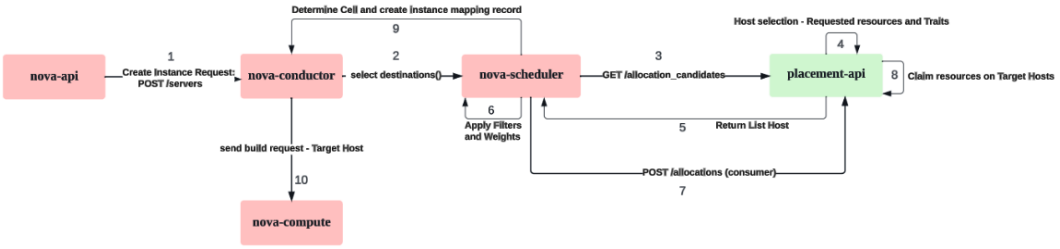


Figure 4.8 – The OpenStack compute placement workflow

When the `nova-api` service receives a user request for a new instance creation, `nova-scheduler` will fire a `GET /allocation_candidates()` request to the `placement-api` service. The `allocation_candidates` method is a collection of a set of resource providers that will allocate

resources based on the attributes encapsulated in the request. A sample API GET request uses the following format as a filter condition to embed the required resources to launch an instance:

```
GET/placement/allocation_candidates?resources=DISK_GB:500,MEMORY_
MB:2048,VCPU:4
```

Upon the previous request, the `placement-api` service will fetch the complying hosts that will accommodate instance flavors: disk capacity of 500 GB, memory size of 2048 MB, and 4 vCPUs.

The previous request can be extended to filter based on resource traits by adding the `&required` query parameter at the end of the request string, so the allocation query will look as follows:

```
GET/placement/allocation_candidates?resources=DISK_GB:500,MEMORY_
MB:2048,VCPU:4 &required=HW_CPU_X86_SVM
```

This way, the `placement-api` service will combine the required resource attributes and an additional trait for hosts that support a CPU with X86_SVM architecture.

Typically, cloud operators can use the OpenStack CLI to deal with placement configuration via the `allocation candidate` command line, as in the following example:

```
# openstack allocation candidate list --resource VCPU=4,
MEMORY_ MB=2048,DISK_GB=500GB --required   HW_CPU_X86_SVM
```

Here's the output:

#	allocation	resource provider	inventory used/capacity	traits
1	VCPU=4,MEMORY_MB=2048,DISK_GB=500	4a4a2eda-44ba-481c-c43a-e3a226a21b11	VCPU=0/512,MEMORY_MB=0/750670,DISK_GB=0/1922	HW_CPU_X86_SVM, HW_CPU_X86_SSE

Figure 4.9 – The placement details of a registered allocation

The main parameters that influence the logic of the subsequent scheduler pre-filtering process lie in `allocation_requests` and `provider_summaries`. The previous objects are returned by the `placement-api` service to the `nova-scheduler` service with two types of objects listed in JSON format with the following data structure:

- `allocation_requests`: A list of resource providers that can satisfy the allocation request:

```
"allocation_requests": [
        {
            "allocations": {
                "ad54422c-b345-5v44-ba23-00cad76e376a": {
                    "resources": {
                        "DISK_GB": 500
                    }
                },
                "652f6729-cd55-4747-91a1-543adfeea2a": {
```

```
                    "resources": {
                        "VCPU": 4,
                        "MEMORY_MB": 2048
                    }
                }
            }
        },
...
```

- `provider_summaries`: The total amount of resources and information usage across all the resource providers that satisfy the allocation request:

```
...
"provider_summaries": {
        "ad54422c-b345-5v44-ba23-00cad76e376a": {
            "resources": {
                "DISK_GB": {
                    "used": 0,
                    "capacity": 2000
                }
            },
            "traits": ["SRG_SHARES_STORE"],
            "parent_provider_uuid": null,
            "root_provider_uuid":
              "ad54422c-b345-5v44-ba23- 00cad76e376a"
        },
        "652f6729-cd55-4747-91a1-543adfeea2a ": {
            "resources": {
                "VCPU": {
                    "used": 0,
                    "capacity": 244
                },
                "MEMORY_MB": {
                    "used": 0,
                    "capacity": 192604
                }
            },
            "traits": ["HW_CPU_X86_SVM", "HW_CPU_X86_SSE2"],
            "parent_provider_uuid": null,
            "root_provider_uuid":
              "652f6729-cd55-4747-91a1-543adfeea2a"
        },
...
```

The JSON `provider_summaries` object output includes the associated traits with a resource provider. The previous output example shows the resource provider for the `52f6729-cd55-4747-91a1-543adfeea2a` compute node UUID, labeled with a trait class supporting the `HW_CPU_X86_SVM` and `HW_CPU_X86_SSE2` CPU architectures.

Filtering

Once the `placement-api` service returns the candidate host list to the `nova-scheduler` service, the latter process runs a variety of filters based on the pre-configured filters in the Nova service. The `/etc/nova/nova.conf` file can be configured to run one or several schedulers by adjusting the `enabled_filters` option.

> **Important note**
>
> The scheduler filters in the `kolla-ansible` repository can be configured in the `filter_scheduler` section, located in the `/kolla-ansible/ansible/roles/nova/templates/nova.conf.j2` file.

Some of the most common filters configured and supported by default in the scheduler are as follows:

- `ComputeFilter`: This returns a list of fully operational hypervisor hosts that should be enabled by default.
- `ComputeCapabilitiesFilter`: This returns a list of hypervisor hosts that are capable of spawning an instance with the requested flavor. For example, extra specifications could include verifying a compute host running a KVM hypervisor.
- `ImagePropertiesFilter`: This returns a list of hypervisor hosts that meet the desired image properties defined in the instance image. Image properties can be added based on the architecture of the hardware or hypervisor type. For example, spawning an instance would require an image that should run on a host supporting KVM as the hypervisor:

```
# openstack image set --property img_hv_type=kvm img-uuid
```

> **Important note**
>
> One of the major updates from the Liberty release is the image properties being passed to the filter for checks. The new supported properties are `hw_architecture` (formerly named `architecture`), which defines the hardware architecture, `img_hv_type` (formerly named `hypervisor_type`), which describes the type of the hypervisor, `img_hv_requested_version` (formerly named `hypervisor_version_requires`), which defines the required version of the hypervisor, and `hw_vm_mode` (formerly named `vm_mode`), which refers to the hypervisor application binary interface.

- `ServerGroupAntiAffinityFilter`: This dispatches instances of the same group in different hypervisor hosts. A common use case is to launch instances running the same workload and requiring an additional level of high availability. To create server groups using an anti-affinity policy, use the `--hint` flag in the Nova command-line tool as follows:

```
# openstack server group create --policy anti-affinity
pp_webgroup
# openstack server create --image "Ubuntu 22.04" --hint
group=122ee342-3345-2234-bac4-1515321e1ebb --flavor 5 "instance
01"
# openstack server create --image "Ubuntu 22.04" --hint
group=122ee342-3345-2234-bac4-1515321e1ebb --flavor 5 "instance
02"
```

  The outcome of the previous command lines will ensure that `instance 01` and `instance 02` are part of the same application server group, `pp_webgroup`, have the `122ee342-3345-2234-bac4-1515321e1ebb` UUID, and run in different compute nodes.

  - `ServerGroupAffinityFilter`: This dispatches instances in the same hypervisor host already running an affinity server group. The affinity scheduler filter is recommended to run workloads that require the highest level of performance and low latency. To create server groups using an affinity policy, use the `--hint` flag in the Nova command-line tool, as follows:

```
# nova server-group-create --policy affinity pp_perfgroup
# nova boot --image "Ubuntu 22.04" –hint group=44ee217-5543-
9871-cda3-1e57522ecde1 --flavor 5 "instance 01"
# nova boot --image "Ubuntu 22.04" –hint group=4ee217-5543-9871-
cda3-1e57522ecde1 --flavor 5 "instance 02"
```

    The outcome of the previous command lines will ensure that `instance 01` and `instance 02`, part of the same application server group, `pp_perfgroup`, having the `144ee217-5543-9871-cda3-1e57522ecde1` UUID, will be running in the same compute node.

OpenStack scheduling supports other filters, with more granular and customized options included in the latest releases of OpenStack. At the time of writing this edition, an exhaustive list of the latest filters and configurations can be found at `https://docs.openstack.org/nova/latest/admin/scheduling.html#filters`.

Weighting

Weighting is the last stage of the scheduling process, wherein the `nova-scheduler` service applies scoring for each filtered candidate host selected from the placement that passed all assigned filters. The compute node with the highest weight will be selected by the scheduler at the final stage. On the surface, the weighting logic might look less complex, but under the hood, additional factors should

be considered. The compute node metrics are periodically monitored and fed to the database to keep track of the host usage and resource claims. The weighting process uses such information as factors and calculates the weights using a normalization function by giving each compute node a value between `0.0` and `1.0`. Compute nodes with the most available resources (as reported by the monitoring) will be assigned a maximum value of `1` and compute nodes with the lowest available resources will have a minimum value of `0`.

The normalized factor is calculated based on the available resources using the following formula:

```
(host_availability_resource - min_value_all)/(max_value_all – min_
value_all)
```

Here, the following applies:

- `host_availability_resource` is the value of the available resource in a compute node
- `min_value_all` is the minimum value of resources across all compute nodes
- `max_value_all` is the maximum value of resources across all compute nodes

The second step of the function is to multiply the normalized value of each compute node for each available resource by the associated weigher multiplier. At the end of the weighting process, the scheduler will sum all normalized weights and select the compute node with the highest score, summarized as follows:

```
Σ (weight(i)*norm_factor)
```

Here, the following applies:

- `weight(i)` is the associated weight multiplier for a resource `i`
- `norm_factor` is the assigned normalized weight factor for each compute node

The following example illustrates the weighting process by calculating the normalized factors and weights for available RAM capacity and CPUs across five compute nodes, respectively. The diagram illustrates the first subset size of nodes (`subset 1`) – the first compiled list of evaluated hosts by the Nova scheduler through the filters – as follows:

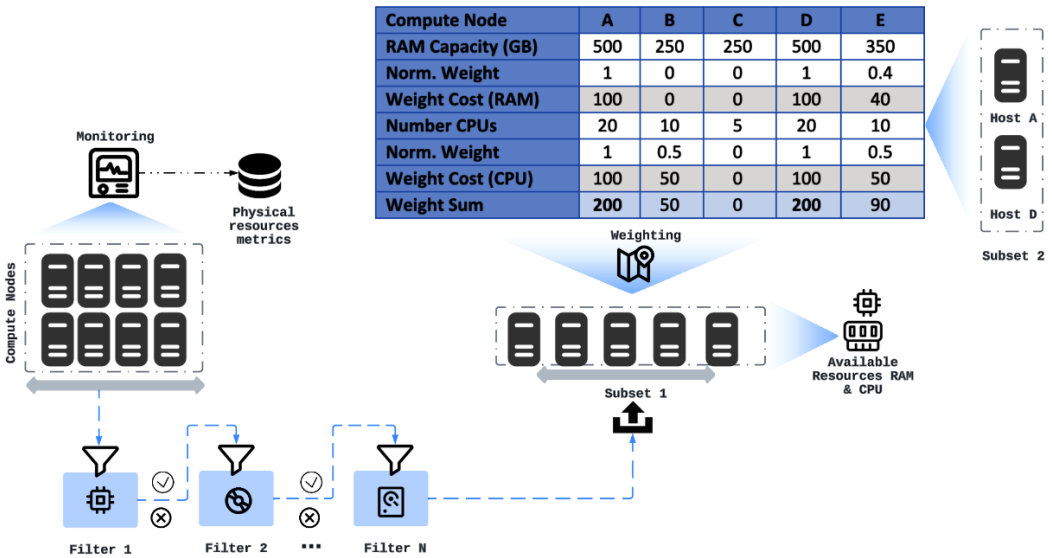


Figure 4.10 – The OpenStack compute weighting mechanism

The scheduler then runs the weighting mechanism and generates the most suitable hypervisor hosts based on the weight scoring as follows:

- Nodes A and D accommodate the highest RAM capacity and are assigned the maximum weight, normalized to `1`. The weight cost is denoted as `100` (1 multiplied by 100).
- Nodes B and C have the lowest amount of RAM capacity and are assigned the minimum weight normalized to `0`. The weight cost is denoted as `0` (0 multiplied by 100).
- Node E is assigned a weight normalized to 0.4 by applying the previous formula to calculate the normalized factor as follows: (350 - 250) / (500 – 250). The weight cost is denoted as 40 (0.4 multiplied by 100)
- Nodes A and D accommodate the highest number of CPUs and are assigned the maximum weight, normalized to `1`. The weight cost is denoted as `100` (1 multiplied by 100)
- Node C has the lowest amount of CPU capacity and is assigned the minimum weight, normalized to `0`. The weight cost is denoted as `0` (0 multiplied by 100)
- Nodes B and E are assigned a weight normalized to 0.5 by applying the previous formula as follows: (10 - 5) / (20 – 10). The weight cost is denoted as 50 (0.5 multiplied by 100)
- The final weighting for each host is calculated by summing up each weight cost of each CPU and RAM resource as follows: host A (100 + 100), host B (0 + 50), host C (0 + 0), host D (100 + 100), and host E (40 + 50)

In this example, both hosts A and D won the weighting bid and the scheduler will simply randomly select one of them from the second node's subset to launch the instance. Resource availability will be reflected in the database and the scheduler will update the weights accordingly on each new instance creation request.

The additional options brought to the filtering and scheduling mechanisms in the compute service can be handy in managing and distributing the workload demand across the compute farm. By default, the scheduling allocates resources for instances by spreading them across the available hosts. In some other cases, dispatching several instances with different flavors and sizes would create a concurrency issue when allocating resources, and hence prevent a subset of instances from being launched. The scheduling weighting policy can be adjusted to use a **stacking** approach. In this case, instances will be allocated to the first filtered host until all of its resources are fully exhausted, before moving on to the next one. A good example of a stacking method is the conjunction of the server affinity configuration and weighting mechanism.

To customize the weighting mechanism in your deployment, make sure to apply the following configuration in the `/etc/nova/nova.conf` file:

1. Enable monitoring on the compute nodes to gather the CPU metrics by setting the `compute_monitors` parameter:

```
compute_monitors = cpu.virt_driver
```

2. Enable the `weights.all_weighers` class:

```
weight_classes = nova.scheduler.weights.all_weighers
```

3. Set the host subset size to `1`. Adjust the scheduler to select the host with the highest weight by setting the host subset size to `1`:

```
host_subset_size = 1
```

4. Adjust the weight multiplier to `1`. This way, the launch of a new instance will be spread evenly between compute nodes:

```
weight_multiplier = 1.0
```

5. Adjust the RAM weight multiplier to `1`. This way, the launch of a new instance will be spread evenly between compute nodes with free memory:

```
ram_weight_multiplier = 1.0
```

Important note

Setting up a weighted multiple as `1.0` will spread the resource allocation across all available filter hosts. If the stacking option is the preferred option, set the associate value of the weight multiplier to a value of `-1`.

6. Add a new multiplier to calculate the weighting host I/O operations. Instruct Nova to use the active hosts evenly with the least I/O CPU metric by setting its weight multiplier to `1.0` as follows:

```
io_ops_weight_multiplier = 1.0
```

7. Restart the `nova-scheduler` and `nova-compute` services in the controller node and each compute node to apply the changes:

```
# service nova-compute restart
# service nova-scheduler restart
```

Important note

The `kolla-ansible` repository does not, by default, provide a low-level configuration for Nova weighting policies. It is possible to adjust the `nova.conf` template file located at `/kolla-ansible/ansible/roles/nova/templates/nova.conf.j2` by adding the configuration lines described previously. It is also recommended to cast an eye on the latest supported `nova.conf` settings, as defined here for the latest release: `https://docs.openstack.org/nova/latest/configuration/sample-config.html`

The OpenStack Bobcat release introduces a new scheduler weigher, `NumInstancesWeigher`, which is handy for dispatching instances based on the number of active instances each compute node is hosting. As demonstrated in the previous example, setting the multiplier value to `1` will enable a `packing` strategy by favoring the host with the most number of active instances. A value of `-1` will favor a spreading strategy by checking the least busy host in terms of active instances. A full list of different weighers can be found at `https://docs.openstack.org/nova/latest/admin/scheduling.html#weights`.

Extending compute for containers

Since 2014, the OpenStack community has been developing additional projects around containerization technology, allowing cloud users to leverage different sets of services to run applications on top of the compute service. The most mature and in-production services are **Magnum** and **Zun**, which we will cover next.

Magnum

The main motivation to run a containerized environment is to run and deploy an application with minimal configuration complexity that's portable and requires the least maintenance effort. Running workloads that are composed of many containers requires a layer of orchestration to manage the logic of the life cycle of different components, as well as inter-communication, in an automated fashion. This is referred to as a container cluster management platform, such as **Mesos**, **Kubernetes**, and **Docker Swarm**, to name a few. There are some well-known public cloud services, such as AWS **Elastic Container**

Service (**ECS**), AWS **Elastic Kubernetes Service** (**EKS**), GCP **Google Kubernetes Engine** (**GKE**), and **Azure Kubernetes Service** (**AKS**). In the OpenStack world, Magnum is the project code name that has been integrated into the OpenStack ecosystem and is referred to as a COE. Like any other OpenStack service, Magnum exposes an API to interact with COEs as first-class resources, such as Kubernetes and Docker Swarm. Containers will be deployed and run on top of the hypervisor nodes. The exposure to different container engines unlocks cloud operators to offer an array of choices for users to select from, as illustrated in the following diagram:

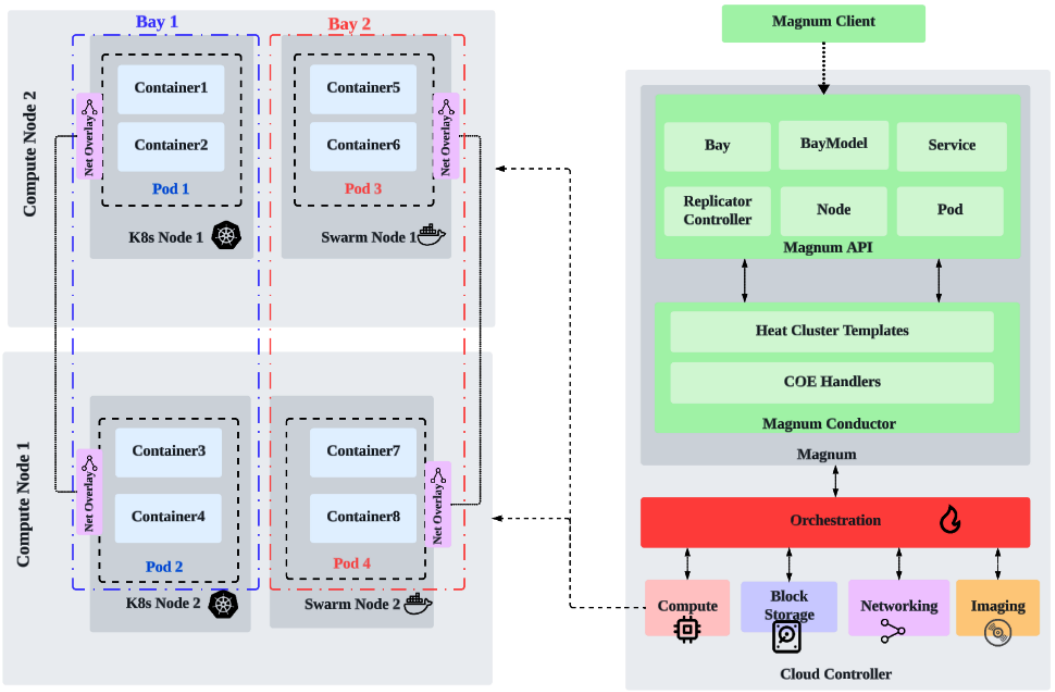


Figure 4.11 – The OpenStack Magnum multi-COE architecture

Magnum refers to each collection of COE nodes and their associated containers using the following terminologies:

- **Bay**: This is a set of nodes running a COE. At the heart of the bay run, Magnum calls the Heat service to deploy the bay. Bays are simply a collection of Nova instances created and orchestrated by Heat using images that target both virtual machines and bare-metal ones.
- **BayModel**: This is a collection of resources that constructs a bay defined in a simple template. A bay can use the same `BayModel` template to run across different COEs.
- **Pod**: This is a group of containers running on the same COE node.
- **Service**: This is an abstraction of an arrangement of bays and access policies.

- **Replication controller**: This is a dedicated process for monitoring, replicating, re-spawning failed containers, and scaling pods across COE nodes.
- **Magnum client**: This is a native client to query the COE that comes with the native **Docker CLI** for the Docker COE and **kubectl** for the Kubernetes COE.

Other emerging keys to Magnum's success are the functions that are well integrated with other OpenStack services, which are listed here:

- **Identity**: This is an integration with Keystone for managing the authentication and authorization of the container cluster platform – for example, creating Kubernetes roles for users to run operational tasks on a cluster via Keystone.
- **Networking**: The Neutron service is leveraged through supported networking drivers – for example, the **Flannel** overlay network is the default one for both the Kubernetes and Swarm COEs, allowing multi-host communication and managing the IP addresses of containers in each bay.
- **Imaging**: A prebuilt Glance image is provided to the Kubernetes and Swarm COEs, which can be configured and used to launch the nodes in the cluster.
- **Storage**: The official supported storage is provided by Cinder as block storage in two forms: ephemeral and persistent storage options. Aside from the Docker storage driver, Cinder also comes with additional backend storage such as **Rexray** as the volume driver. At the time of writing this, Cinder only supports the Rexray driver for Swarm and the default Cinder for Kubernetes.

Other main highlights of Magnum that have been initiated or are in progress can be summarized as follows:

- **Security posture**: The Magnum architecture is secure by design. Each bay is isolated from others, providing security for the workloads they host. Additionally, Magnum enables a multi-tenancy layout by preventing bays from being shared between OpenStack tenants.
- **Scalability**: The scaling of the cluster nodes is performed automatically by adjusting the Heat template attributes or manually via the OpenStack COE CLI. Note that container scaling will depend on the cluster container configuration as per the COE.
- **High availability**: At the time of writing this, the Magnum high availability criteria supports only one default Availability Zone. The group of compute nodes will be spawned in the same logical Availability Zone. The OpenStack community is considering enlarging the span of Magnum cluster nodes across more multi-Availability Zones in the next releases.

Since the Liberty release, the Magnum service has received a lot of credit due to its exposure to an array of COEs and maturity, which can be seen in dozens of production workloads. The service has enabled engineering and architect teams to migrate to microservice-based architecture without the need to redesign and re-integrate the existing container toolsets. Other enhancements have been captured and can be followed by checking the release updates at `https://docs.openstack.org/magnum/latest/`.

The Magnum service is not enabled by default in the current `kolla-ansible` infrastructure code repository. Setting `enable_magnum` to `yes` in the `/kolla-ansible/etc/kolla/globals.yml` file will enable the Magnum API service. Make sure to update the inventory file created in the previous chapter, `/ansible/inventory/multi_packtpub`, by adding the Magnum services, including the `magnum-api` and `magnum-conductor` services, in the controller node as in the following configuration stanza code:

```
…
[magnum:children]
control
[magnum-api:children]
magnum
[magnum-conductor:children]
magnum
…
```

Important note

More Magnum configuration options are listed in the Ansible playbook, under the `/kolla-ansible/ansible/roles/magnum/default/main.yml` file. Additionally, Horizon supports the Magnum panel in the dashboard, which is disabled by default and can be added by setting `enable_horizon_magnum` to `yes` in the `/kolla-ansible/etc/kolla/globals.yml` file.

Once both files are updated, run the CI/CD pipeline to roll out the Magnum Ansible playbooks and provide the Magnum service containers as part of the control plane in the OpenStack environment.

Zun

Beyond the Magnum service and its multitude of container orchestration support options, there is the Zun service – a very simple service to run containers quickly in OpenStack without the need to manage or deploy a container orchestration environment based on Kubernetes Pods or Docker Swarm clusters. You can simply fire a Zun command line to create any type of container for any Docker repository and run your workload. To resolve any confusion between Magnum and Zun use cases, the latter is dedicated to managing and operating containers through OpenStack directly without the need to interface with an additional orchestration toolset layer of a COE such as Kubernetes or Swarm, unlike the Magnum service. Another potential source of confusion is the difference between Zun and the former OpenStack container originator, the `nova-docker` service. While `nova-docker` relies on the Nova API service to manage containers, Zun is independent of the Nova API and leverages its fully featured API interface to create and operate containers. The Zun service is well integrated with a number of versatile OpenStack services, including Nova, Glance, Keystone, Neutron, Cinder, Ceilometer, and Magnum, as illustrated in the following diagram:

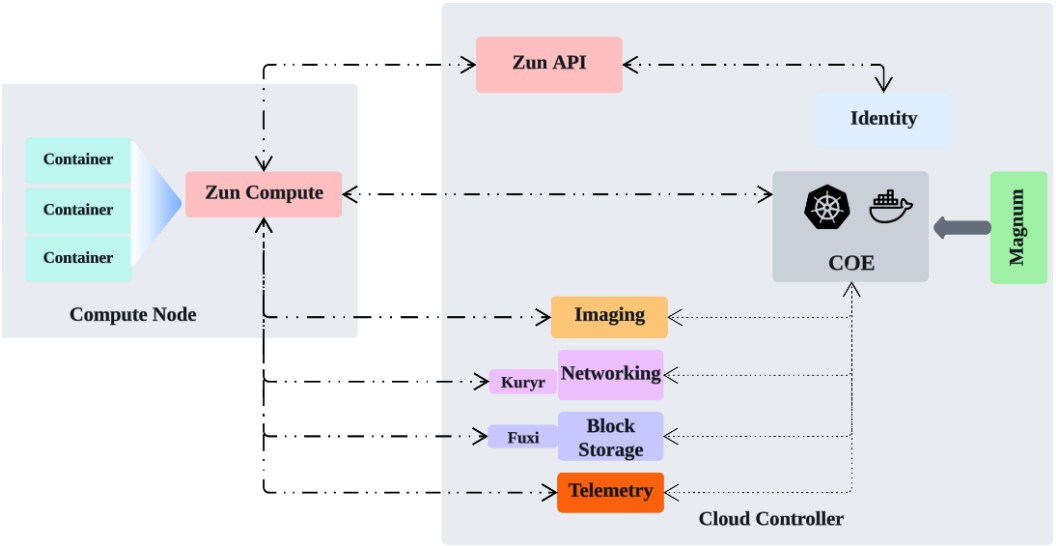


Figure 4.12 – The OpenStack Zun compute integration

Figure 4.12 exposes two additional services that Zun compute uses and runs in tandem with Neutron and Cinder, respectively summarized as follows:

- **Kuryr**: This integrates with Neutron to provide advanced networking features for containers through **libnetwork**.
- **Fuxi**: This leverages the Docker API to manage volumes backed by Cinder and Manila storage services.

The Zun service is not enabled by default in the current `kolla-ansible` infrastructure code repository. To enable the service, the following settings should be adjusted in the `/kolla-ansible/etc/kolla/globals.yml` file as follows:

```
...
enable_zun: "yes"
enable_kuryr: "yes"
enable_etcd: "yes"
docker_configure_for_zun: "yes"
containerd_configure_for_zun: "yes"
...
```

Make sure to update the inventory file created in the previous chapter, `/ansible/inventory/multi_packtpub`, by adding the Zun services, including `zun-api` and optionally `zun-wsproxy`, as part of the control plane as follows:

```
…
[zun:children]
control
[zun-api:children]
zun

[zun-wsproxy:children]
zun
```

The `zun-compute` and optionally `zun-cni-daemon` services can be added as part of the compute node as depicted here:

```
...
[zun-compute:children]
compute
[zun-cni-daemon:children]
compute
…
```

> **Important note**
>
> More Zun configuration options are listed in the Ansible playbook under the `/kolla-ansible/ansible/roles/zun/default/main.yml` file. Additionally, Horizon supports the Magnum panel in the dashboard, which is disabled by default and can be added by setting `enable_horizon_zun` to `yes` in the `/kolla-ansible/etc/kolla/globals.yml` file.

Once both files are updated, run the CI/CD pipeline to roll out the Magnum Ansible playbooks. The Zun service and its associated processes will be launched in containers as part of the control plane in the OpenStack environment.

> **Important note**
>
> Make sure to double-check that the `etcd` daemon is running on the controller node. Similarly, ensure that the Docker runtime and `kuryr-libnetwork` are properly installed in the compute node.

Check that the Zun service can be performed through the Zun service list command line:

```
$ openstack appcontainer service list
```

Zun containers should be created in the compute nodes. To list the registered compute nodes for Zun, use the following command line:

```
$ openstack appcontainer create --name container_pp_zun cirros
```

Creating containers using the Zun CLI is straightforward. It is possible to use, by default, **Docker Hub** to pull images and rapidly run a container in the designated compute node. The following example command line creates a Zun container using a CirrOS image:

```
$ zun create --name container_pp_zun cirros
```

An amazing hands-on feature brought by Zun is the flexibility to manage a container's life cycle by using the native Docker API or the Zun API itself. For example, attaching a terminal to a running container can be performed by running `openstack appcontainer attach <container_name>` in a similar way to firing a `docker attach` command line.

Expanding the compute farm

Managing the compute fleet hypervisors through `kolla-ansible` to spawn instances or containers is straightforward. After collecting your compute requirements, hardware capacity, types of workloads, and scheduling strategies, adding a compute node can be performed on the fly without incurring a complex operation overhead. The main requirements that should be considered are the listing of compute services – mainly, `nova-compute` and a networking plugin agent to provide connectivity with the control plane and sync with `nova-scheduler` for each resource allocation. The previous section demonstrated that additional components can be part of the compute node, including the Zun compute service, if you're planning to offer an easy and quick container playground to end users. Another aspect of managing compute nodes from code through `kolla-ansible` is listing the available configuration options of the compute service, such as overcommitment ratios for CPU and memory, covered in *Chapter 1*, *Revisiting OpenStack – Design Considerations*. As you might deploy several compute nodes with different configurations, `kolla-ansible` provides ways to customize compute service configuration and override configuration with the global configuration defined in the `/etc/kolla/globals.yml` file. The following example will instruct Kolla to add a new compute host named `cn02.os.packtpub` (24 CPUs and 255 GB of RAM) with custom configuration supporting QEMU as a hypervisor, and a CPU and RAM overcommitment ratio of `8.0` and `4.0`, respectively:

1. First, let's start by populating the hostname in the inventory file:

```
…
[compute]
…
cn02.os.packtpub
```

2. Create a new `nova.conf` file under `/etc/kolla/config/nova/cn02.os.packtpub/`. Kolla looks for any possible config file under `/etc/kolla/`. If one is found, Kolla will override and merge the global one with the custom settings provided in a separate file. Note that you will need to create the path directories in the `kolla-ansible` repository, including the hostname of the compute node, following the `/etc/kolla/config/nova/COMPUTE_HOSTNAME/nova.conf` format, wherein `COMPUTE_HOSTNAME` is the hostname of the compute node.
3. Edit the newly created `nova.conf` file by adding the CPU and RAM allocation ratio values of `8.0` and `4.0`, respectively, in a new section:

```
[DEFAULT]
cpu_allocation_ratio = 8.0
ram_allocation_ratio = 4.0
```

4. Add to the same file the desired `qemu` hypervisor to run on the new compute node:

```
...
[libvirt]
virt_type=qemu
cpu_mode = none
```

5. It is possible to overwrite the default scheduler by forcing a set of filters and using the `ComputeCapabilitiesFilter` filter to use any host with total available RAM for users of more than 250,000 MB:

```
...
[filter_scheduler]
enabled_filters = ComputeFilter,ComputeCapabilitiesFilter,
ImagePropertiesFilter,ServerGroupAntiAffinityFilter,
ServerGroupAffinityFilter
```

6. Optionally, you can use the filtering strategy for RAM criteria by stacking instances in the same host with a defined flavor. That way, you can make sure that the compute node will exactly accommodate a defined number of instances with specific flavors that you can predict in advance. By doing so, the 250 GB of RAM can be used fully without leaving smaller slots that would otherwise lay unused. The way to go in this case is the stacking strategy, by defining a negative weight multiplier in the same `custom nova.conf` file as follows:

```
...
weight_classes nova.scheduler.weights.all_weighers
ram_weight_multiplier -1.0
```

7. Using your CI/CD pipeline, trigger the run job to deploy the code in the staging environment, as demonstrated in the previous chapters, before merging to the production branch. The deployment stage should provision the new additional Kolla containers in the new compute node and start syncing with the control plane.

8. An example of resource allocation inspired by the filtering strategy covered in this chapter and *Chapter 1*, *Revisiting OpenStack – Design Considerations* is to dispatch a specific flavor of instances to a dedicated compute node. For example, a cloud operator would use the scheduler filter described in *step 5* to provision instances only with medium-sized allocation requests. For that to take effect, run the following command line in the controller node:

```
$ openstack flavor set m1.medium --property total_usable_ram_
mb=">= 250000"
```

The enabled `ComputeCapabilitiesFilter` filter should match any compute node attributes with a flavor, as defined against its usable amount of memory attribute for a specific instance flavor. If the available amount of RAM in the first compute node is below 250, a new instance request of the `m1.medium` size will likely be dispatched by the scheduler on the new compute node that has freshly joined the compute node farm.

By the end of the previous steps, a new compute node should join the OpenStack compute cluster with advanced scheduling configured based on filter and weighting mechanisms. Any new compute request with the `m1.medium` flavor will be launched in the new compute node if the amount of RAM available is at least 250 GB.

Summary

In this chapter, we looked at some powerful capabilities and additional features brought by the Nova service. You should be acquainted with the vast array of choices given by the latest updates in the compute services – from the variety of supported hypervisor technologies, to compute and workload segregation, to the art of filtering. Scheduling in OpenStack is a key element that cloud operators should pay special attention to when dealing with growing infrastructure and high demand for resources. Although we did not cover all possible scenarios, the filtering mechanism was introduced in the early days of the Nova project, and this chapter demonstrated that an extension to deploy compute power beyond a limited perimeter has since become possible. Thanks to the new, mature version of Nova CellV2, the host aggregation and Availability Zone concepts enable massive deployments if planned and designed properly, as we learned. At the end of the chapter, both Magnum and Zun were covered as popular container projects within the OpenStack ecosystem. We learned that cloud users can run native container applications driven by Magnum as a COE and Zun as a simple container API that's well integrated with different core services around the OpenStack ecosystem. Furthermore, we learned that adopting microservice prototypes and running native cloud applications is no longer a challenge for an OpenStack user. With the continuous development of those services, enterprises can adopt a hybrid environment between private and public clouds not only due to the portability and flexibility of the containers but also because OpenStack has proven its capability to be in the container race with the other public cloud players. We discussed this in this chapter.

Nova is not the only service that has seen some updates. Storage services including file share, block, and object storage have been empowered with additional features, which we will cover in the next chapter.

5

OpenStack Storage – Block, Object, and File Shares

"Perseverance, secret of all triumphs."

– Victor Hugo

The wide adoption of OpenStack services has increased user demand and, in turn, its array of capabilities, as demonstrated in the previous chapter with the compute service. This feature variation has also extended the OpenStack storage offering. Users running workloads on top of an OpenStack environment would require more than one storage type in addition to a robust and reliable storage solution out of the box. The spread of the **software-defined storage** (**SDS**) approach has enabled the OpenStack community to adopt more storage projects in which data storage is abstracted from physical storage. Storage systems can be built on top of commodity hardware, but that would require a good understanding of each storage project's purpose and the architecture behind it before exposing the service to cloud users so that they can run workloads.

In this chapter, an overwhelming array of storage services in OpenStack will be discussed and enabled in our existing OpenStack environment. Throughout this chapter, we will cover the following topics:

- Reviewing the Cinder Block Storage service in OpenStack
- Expanding the Block Storage setup by configuring the three most used driver backends, using Kolla-Ansible for LVM, NFS, and Ceph
- Uncovering the state of scheduling in Cinder and enabling a weighing mechanism in the existing cluster
- Discussing the Swift Object Storage service in OpenStack
- Deploying an Object Storage cluster using kolla-ansible
- Exploring the latest updates on the file-sharing service, Manila
- Integrating the Manila service and deploying it using `kolla-ansible`

Defining Block Storage – Cinder

Cinder provides persistent storage for instances. The Block Storage service has been extensively developed through each OpenStack release to support more features and vendors' backend drivers, allowing for a variety of storage devices to be used. Most of the used storage backends include **logical volume management** (**LVM**), IBM storage drivers, NetApp, and Dell storage. Within the latest OpenStack releases, new backend drivers have been supported, such as Yadro, TOYOU, and Pure Storage FlashArray.

The storage device is reached by Cinder via its management access API and grants direct access to the instances to access and attach volumes. A volume attached to an instance is shown as an additional hard drive can be partitioned and mounted onto the filesystem of a virtual machine. The volumes are generally accessed by the defined storage path, which could use iSCSI, NFS, and Fiber Channels to do so. It is important to bear in mind that Block Storage is persistent, which means it grants no data loss when an instance is terminated, unlike an ephemeral disk.

In our multi-node environment layout, the latest novelty of the Cinder service can be illustrated as follows:

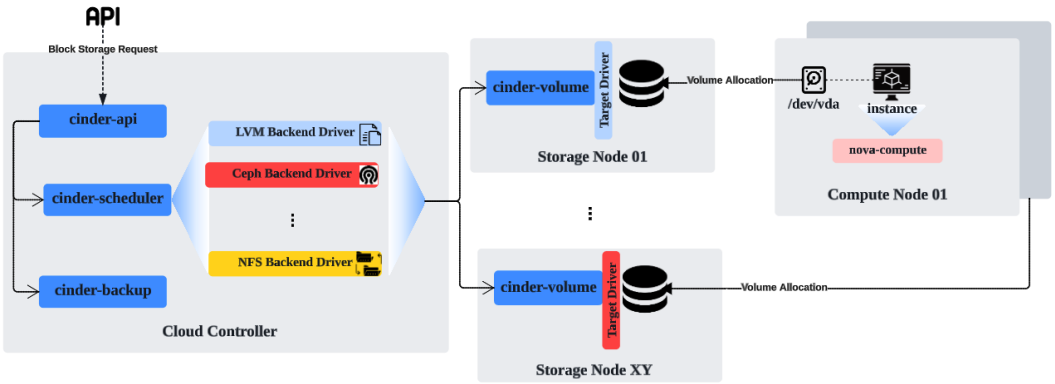


Figure 5.1 – Block Storage core architecture

The core Cinder components can be summarized as follows:

- `cinder-api`: Processes block storage REST API requests and responses
- `cinder-scheduler`: Like `nova-scheduler` in the Compute service, it redirects after filtering step requests to the appropriate `cinder-volume` server that will handle and provision the requested volume
- `cinder-volume`: Acts as a volume manager and runs in each storage node
- `cinder-backup`: Enables volume backups to different storage systems

The default OpenStack deployment would come with a Block Storage backend based on LVM. The other very common use case of the backend is the **Network File System** (**NFS**), which leverages existing shared storage. Here, the only thing that you need to do is configure the Cinder service so that it can use the NFS driver. It is important to keep up to date with the latest supported backend drivers by checking out the Cinder driver features matrix, available here: `https://docs.openstack.org/cinder/latest/reference/support-matrix.html`.

The driver's storage backends should support the minimum required volume functions, such as volume attachment, detachment, creation, deletion, extension, migration, and the creation of an image from a volume. The drivers should also provide snapshot management features such as creation, deletion, and volume creation from a snapshot or a cloned volume. A more extensive list of features can be considered optional within the latest OpenStack releases, depending on each backend driver, such as thin volume provisioning, live migration, multi-attach support, and **quality of service** (**QoS**). Make sure you scan the existing storage backends and reflect the features supported in the matrix before moving to a full-block storage deployment.

In addition to the aforementioned drivers' capabilities, Cinder supports a variety of configurable drivers for different storage vendors, such as NetApp, IBM, VMware, Dell, and Synology. Bear in mind that not all backend products support all Cinder driver features. On the other hand, the aforementioned Cinder operations are considered *must-haves* when including a new driver storage backend in the Cinder code.

Security in Block Storage has been also tweaked by elaborating more on encryption in transit and at rest. The Antelope release supports most of the Cinder driver's TLS, which was a blocker in old releases for security teams when they dealt with traffic initiated from different service endpoints, as well as end users. Encryption at rest has also reached a milestone, allowing data in volumes to be encrypted using the Cinder API.

Another recent Cinder addition is the Block Storage backup capability. It has become easier to simply fire a command line and instruct Cinder to either make a full backup of the volume or an incremental one if the backup exists.

Expanding storage with multiple backends

Our initial deployment highlighted the usage of a dedicated node to deal with storage and volume management in the OpenStack environment. In the next section, we will expand the initial deployment highlighted in *Chapter 2*, *Kicking Off the OpenStack Setup – The Right Way (DevSecOps)*. Depending on which storage backends will be used, additional nodes will be dedicated to hosting Cinder volumes as part of the data plane. Cloud controller nodes host the Cinder API and scheduling services.

Deploying with LVM

OpenStack's default block storage backend is LVM. The `cinder-volume` service uses the iSCSI target to operate and manage access to the logical volumes in the storage node.

On the storage nodes, we will create an LVM physical volume group. In this example, we assume that `/dev/sdb` is an available block device on the target storage node:

```
$ pvcreate /dev/sdb
```

> **Important note**
>
> Depending on the operating system, disk partitions might have different name listings. You can check these out by using the `fdisk` command line.

Next, create a new volume group named `cinder-volumes`:

```
$ vgcreate -f cinder-volumes /dev/sdb
```

In the `kolla-ansible` code repository, add the new storage hostname to the inventory file located under `/ansible/inventory`, named `multi_packtpub_prod`. In this example, we will use the `storage02.os` storage node:

```
…
[storage]
storage02.os.packtpub
[cinder-volume:children]
storage
…
```

Enable the LVM backend in the `/etc/kolla/globals.yml` file:

```
…
enable_cinder_backend_lvm: "yes"
…
```

> **Important note**
>
> Once the LVM backend has been enabled, `kolla-ansible` activates the iSCSI module that comes with the Ubuntu operating system and is configured by default in the Cinder role, located in the `/kolla-ansible/ansible/roles/cinder/defaults/main.yml` file. For CentOS distributions, make sure that the `LioAdm iSCSI` target helper is installed beforehand.

Configure the name of the Cinder volume group created previously in the `/etc/kolla/globals.yml` file:

```
…
cinder_volume_group: "cinder-volumes"
…
```

Run the job CI/CD pipeline in your staging environment before promoting the additional node to the production one. Once deployed, a new kolla container running `cinder-volume` should be running in the target storage node.

Deploying with NFS

NFS is another commonly used storage backend. Within existing NFS storage, cloud operators can smoothly manage volumes via the NFS protocol. Created shares will be available and mounted by the `cinder-volume` service to compute nodes. The Cinder NFS driver enables access to file images on the shared NFS server and takes care of mapping the files to instances as block storage.

In the following example, an NFS server named `nfs-host-pp` with the `nfs-host-pp:/nfs/share/cinder` share file path is available.

Create a share file in the target storage node by running the following command:

```
$ echo "nfs-host-pp:/nfs/share/cinder" > /etc/cinder/nfs_share
```

Make sure the NFS client is installed in both the compute and storage nodes. You can do this by using Ansible or the following command-line tool:

```
$ apt-get install nfs-common
```

Enable `cinder-volume` in the Block Storage node to access the share file by adjusting the file permissions:

```
$ chown root:cinder /etc/cinder/nfs_share
$ chmod 640 /etc/cinder/nfs_share
```

In the NFS backend, make sure you specify the path of the share directory on the storage node, the storage network, and the access modes, including share, read-write, and non-root remote user access in the `/etc/exports` file:

```
/nfs/share/cinder 10.30.0.0/24(rw,sync,no_root_squash)
```

For custom NFS deployment settings, you can create a new configuration file under `/etc/kolla/config` that will take precedence when running `kolla-ansible`. The following example configuration file, named `nfs_shares`, will include a target storage node, `storage02.os`, with custom NFS-mounted file shares:

```
…
storage02.os:/nfs/share/cinder
…
```

Enable the NFS backend in the `/etc/kolla/globals.yml` file:

```
…
enable_cinder_backend_nfs: "yes"
…
```

Running the job CI/CD pipeline with a newly configured NFS share will restart the `cinder-volume` service in the target storage node. Once deployed and running, the new NFS share directory should be mounted and visible in the storage node. On the storage node, run the following command line to validate the mapped NFS mount:

```
 # mount | grep nfs
nfs-host-pp:/nfs/share/cinder on /etc/cinder/nfs_share
223af296419e436d9142928374d8e57e
type nfs4
(rw,relatime,vers=4.1,rsize=81921,wsize=81921,namlen=255,hard,proto
=tcp,port=0,clientaddr=10.30.255.1,local_lock=none,addr=10.30.255.1
)
```

Each NFS share uses a mount point under `/etc/cinder/nfs_share` with a hashed directory, `223af296419e436d9142928374d8e57e`, that will host any newly created volume. You can check this by running the following command line:

```
# openstack volume create --size 10 nfs-volume
```

The new volume should be located under the hashed directory of the NFS mount point:

```
# ls /etc/cinder/nfs_share/223af296419e436d9142928374d8e57e
volume-b2be4de1-6e42-e3de-15e8-a72b60ca91ef
```

NFS storage is quite a popular storage backend in OpenStack. Another type of backend is based on SDS, which will be covered in the next section.

Deploying with Ceph

Ceph's SDS system gained significant success since its early days. Ceph is designed to scale massively to exabytes of storage pools that can run on commodity x86 hardware architecture. On top of that, Ceph interfaces support most storage types, including object, block, and file shares.

The OpenStack Foundation has committed to ensuring the integration of Ceph, and it can be incorporated and utilized within an existing OpenStack environment. At its core is the **Reliable Autonomic Distributed Object Store** (**RADOS**), which deals with the distribution, replication, and management of objects within the Ceph storage cluster. Based on Ceph's RADOS design, operators can define Ceph storage pools to be used as block storage and provide volumes to OpenStack instances. This can be achieved by another mature Cinder backend driver known as a **RADOS Block Device** (**RBD**).

Important note

Ceph's official documentation provides a complete description of the Ceph cluster setup using cephadm, available at `https://docs.ceph.com/en/latest/cephadm/install/`.

Ceph defines a set of core logical components:

- **Object Storage Devices** (**OSDs**): These correspond to the physical disks within a filesystem, such as **XFS** or **Btrfs**.
- **Monitor daemon server** (**MON**): Watches the state of data consistency and other metrics in each OSD node.
- **Pool**: Provides mapping for stored objects in an OSD.
- **Placement groups** (**PGs**): Maps of each object stored and OSDs. Replicates objects across multiple OSDs within a pool.

If you have an existing Ceph cluster or are willing to deploy a new one, the following walk-through will guide you through the required steps for a basic Ceph integration in OpenStack. In the following example, a dedicated cluster of three Ceph nodes runs `ceph-osd`. Optionally, `ceph-mon` will be running in the Cloud Controller node, as shown in the following figure:

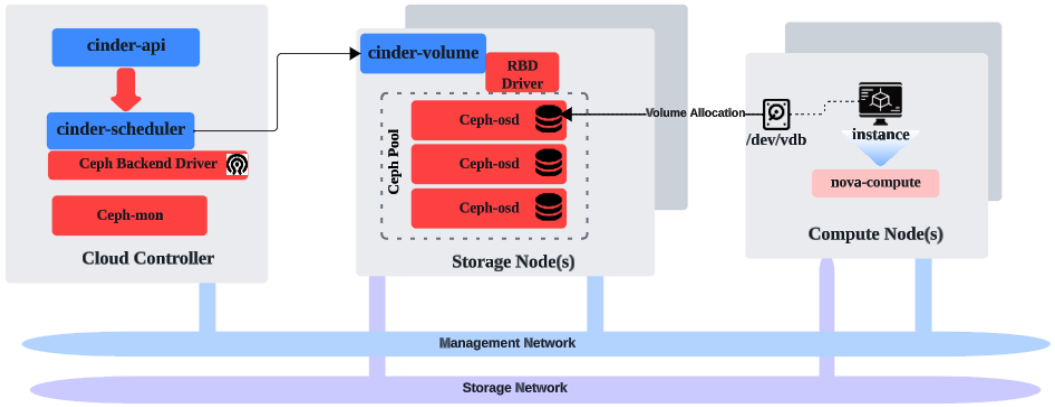


Figure 5.2 – Ceph storage integration in OpenStack

Make sure you have the Ceph client and Python `rbd` library packages installed on each controller, compute, and storage node. Run the following command line if the Ceph client hasn't been installed:

```
$ apt-get install ceph-common python-rbd
```

On a Ceph node, create a storage pool to be used by Cinder:

```
$ ceph osd pool create cinder-volumes 256
```

Initialize the created pool using the `rbd` command-line interface:

```
$ rbd pool init cinder-volumes
```

Create a Ceph user and keyring for Cinder to access the created `cinder-volumes` pool:

```
$ ceph auth get-or-create client.cinder mon 'profile rbd' osd 'profile
rbd pool=cinder-volumes' mgr 'profile rbd pool=volumes' > ceph.client.
cinder.keyring
[client.cinder]
key = QA7sduw73dx83ks02210dj9Lmfj00sdju3ndoy==
```

Copy the created keyring to the OpenStack deployer, storage, and compute nodes:

```
# ceph auth get-or-create client.cinder | ssh storage02.os sudo tee /
etc/kolla/config/cinder/cinder-volume/ceph.client.cinder.keyring
# ceph auth get-or-create client.cinder | ssh cc01.os sudo tee /etc/
ceph/ceph.client.cinder.keyring
# ceph auth get-or-create client.cinder | ssh cn01.os sudo tee /etc/
ceph/ceph.client.cinder.keyring
```

Verify that the Cinder client can access the Ceph cluster by running the following command line:

```
# ceph -s --name client.cinder --keyring /etc/ceph/ceph.client.cinder.
keyring
```

Enable the Ceph backend in the `/etc/kolla/globals.yml` file:

```
…
cinder_backend_ceph: "yes"
…
```

Additionally, make sure you specify the Ceph cinder user, the name of the pool, and the keyring in the `globals.yml` file:

```
…
ceph_cinder_keyring: "ceph.client.cinder.keyring"
ceph_cinder_user: "cinder"
ceph_cinder_pool_name: "cinder-volumes"
…
```

Set the RBD Cinder UUID secret from the `/etc/kolla/passwords.yml` file:

```
# cat /etc/kolla/passwords.yml |grep cinder_rbd_secret_uuid cinder_
rbd_secret_uuid: 612271b2-4537-3e21-65fa-a2e2523d3e421
```

From a Ceph node, copy the `ceph.conf` file to the custom `/etc/kolla/config/cinder` directory. An example of a Ceph configuration extract is shown here:

```
[DEFAULT]
enabled_backends=rbd-1

[rbd-1]
rbd_ceph_conf=/etc/ceph/ceph.conf
rbd_user=cinder
backend_host=rbd:volumes
rbd_pool=cinder-volumes
volume_backend_name=rbd-1
volume_driver=cinder.volume.drivers.rbd.RBDDriver
rbd_secret_uuid = 612271b2-4537-3e21-65fa-a2e2523d3e421
[client]
rbd_cache = True
rbd_cache_size = 335544320
auth_cluster_required = cephx
auth_service_required = cephx
auth_client_required = cephx
...
fsid = 83baa63b-c421-480a-be24-0e2c59a70e17
mon_host = 10.30.0.22
mon_initial_members = ceph-1

cluster_network = 10.30.0.0/24
...
```

Roll the new changes by running the job CI/CD pipeline. The `kolla-ansible` playbook will restart the `cinder-volume` container with a new configuration. The storage node should have a Ceph RBD driver enabled and working access to the Ceph cluster. A new volume request should trigger `cinder-volume` to create the volume in the `cinder-volume` Ceph pool.

OpenStack operators could leverage the usage of multiple storage backends simultaneously. To enable such a capability efficiently, Cinder supports a special scheduling mechanism. We'll explore this in the next section.

Storage filtering and scheduling

As demonstrated in the previous section, a variety of storage backends are supported by Cinder. The block storage requests and their associated storage backend types are handled by `cinder-scheduler`. Similar to the Compute scheduler service, the Block Storage scheduler service takes care of assigning each volume to be created to a specific backend available in the storage pool. The Cinder scheduler uses filter policies to select the best backend fit for a new volume creation request. A few filtering criteria will be evaluated based on the storage information capabilities, such as the state, available space, and drive state. Afterward, `cinder-scheduler` uses a weighing mechanism that assigns each filtered backend a weight and sorts them out, as illustrated in the following figure:

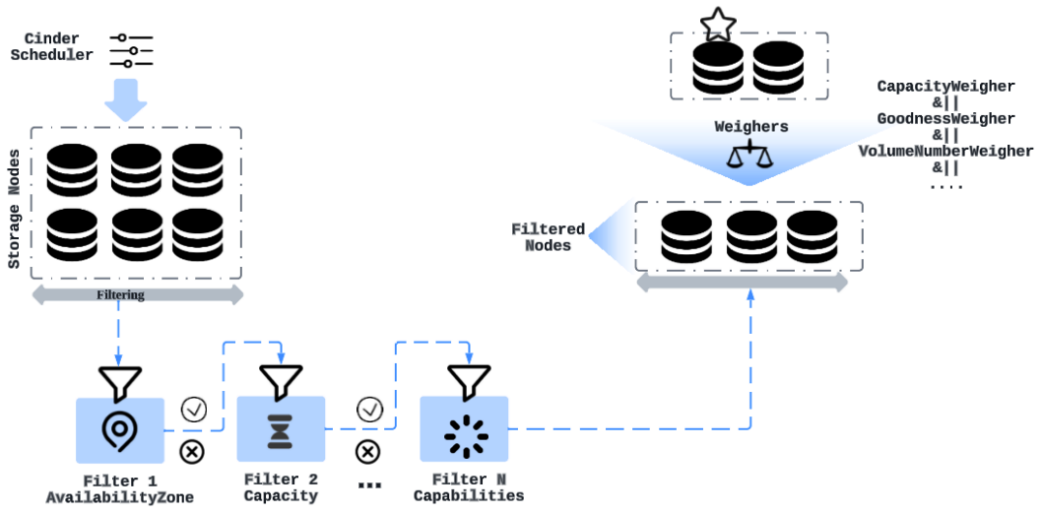


Figure 5.3 – Cinder's scheduling and filtering mechanism

The scheduler periodically checks the state of the listed backends and continuously updates the storage backend candidate list.

By default, Cinder uses `CapacityFilter` and, for weighing, `CapacityWeigher`, which filters the volume backends based on the storage size utilization. The latest Cinder scheduler filters and weighers, at least up until the Bobcat release, can be found at `https://docs.openstack.org/cinder/latest/configuration/block-storage/scheduler-filters.html` and `https://docs.openstack.org/cinder/latest/configuration/block-storage/scheduler-weights.html`, respectively. The following Cinder scheduler weighers list has changed over OpenStack releases, which lists the most adopted weighers in large deployments:

- `CapacityWeigher`: Storage backend(s) are assigned the highest weight based on the most available and free storage capacity.

- `VolumeNumberWeigher`: Balances the volume allocation between different storage backends filtered by the scheduler. This is the best option to spread multiple volume creation requests evenly across different backends that share the same name.
- `GoodnessWeigher`: A more fine-grained weigher that places volumes based on specific storage properties. It uses a formula that evaluates a storage backend, using a rating function, denoted as `goodness_function`, with the `"(property_rule )?challengeVal1 : challengeVal2"` format. The attribution of weights varies between `0` (the lowest) and `100` (the highest), based on the evaluation of `property_rule`, as follows:
  - If `property_rule` returns `true`, then the storage backend is assigned the weight of the `challengeVal1` value
  - If `property_rule` returns `false`, then the storage backend is assigned the weight of the `challengeVal2` value

The following step-by-step configuration will demonstrate the usage of Cinder scheduling, by enabling default filters and using `GoodWeigher` to choose an ideal storage backend for each volume request. In this setup, two of the storage backends we configured in previous sections will be used for LVM and Ceph, respectively. We can continue to add a custom scheduling configuration by creating a new file, `/etc/kolla/config/cinder/cinder.conf`. Running `kolla-ansible` will merge any configuration file under the `/etc/kolla/config/cinder` directory with the one in the main template defined in the `/kolla-ansible/ansible/roles/cinder/templates/cinder.conf.j2` file. The new file will be configured with the following settings:

1. Enable the list of schedulers in a new section:

```
[default]
scheduler_default_filters =
DriverFilter,CapacityFilter,CapabilitiesFilter
```

2. Add `GoodnessWeigher` as a main weigher mechanism, as follows:

```
scheduler_default_weighers = GoodnessWeigher
```

3. For each backend section, add `goodness_function` to define a rating based on the storage provider's capability. The next simple configuration evaluates the volume size and checks whether both backends' utilization of `lvm-1` and `rbd` is less than 50%. If so, `50` and `80` weights will be assigned, respectively. In this case, `rbd` is the best candidate to place the volume. If the backend utilization on `lvm-1` is greater than 50% and less than 50% for `rbd`, the weigher will assign

80 to rbd and a weight of 30 to lvm-1. The last case of weight logic is when both backends' utilization is more than 50% – in this case, lvm-1 will be selected with 30 compared to rbd, which is assigned a weight of 20:

```
[lvm-1]
...
goodness_function = "(capabilities.utilization < 50.0)?50:30"
...
[rbd]
...
goodness _function = "(capabilities.utilization < 50.0)?80:20"
```

4. Run the same job's CI/CD pipeline, and make sure that kolla-ansible will restart the cinder-volume container with a new configuration. The cinder.conf file should have an extended configuration list of the configured schedulers and weighers.

Cinder provides constructs to request volumes with a specific storage backend, using volume types. The following example shows how to define two volume types for LVM and Ceph, based on each backend name to be used when creating volumes:

```
$ openstack volume type create lvm_standard
$ openstack volume type set lvm_standard --property volume_backend_
name=lvm-1
$ openstack volume type create rbd_large
$ openstack volume type set rbd_large --property volume_backend_
name=rbd-1
```

Users should be able to create volumes transparently based on storage types, as defined in the preceding command lines. The storage pool offer can be classified as large block storage with replication capabilities, defined as rbd_large, and a second standard storage capability for general purposes, defined as lvm_standard:

```
$ openstack volume create --size 10 --type lvm_standard general_volume
$ openstack volume create --size 50 --type rbd_large large_volume
```

In this scenario, the Cinder scheduler will assign each volume request to the designated backend storage – lvm-1 and rbd, respectively. In other cases, two or more storage backends can share the same volume backend name, volume_backend_name – in this case, the scheduler will promote the storage backend type based on its scheduling and weighing filters.

With the integration of various storage solutions and advanced scheduling mechanisms, cloud operators can take advantage of an array of features in each storage solution, managed through a single Cinder interface. Besides the Block Storage option, OpenStack offers another storage service focused on objects, code named Swift. We will cover this in the next section.

Revisiting object storage – Swift

Swift is a service tailored to store large volumes of unstructured object data, such as text or binary data. It enables OpenStack operators to build distributed object storage systems by using commodity storage. The hallmark of the Swift service is the way it is designed, which ensures data availability and durability. Several workloads that can run in an OpenStack cloud environment, such as web applications, can leverage object storage by enjoying the advantages of this simple service's usage and management, as well as its transparent data replication and horizontal scaling.

Revisiting Swift

Swift's logical core components consist mainly of the following:

- **Account server**: Represents a namespace for a list of containers associated with a Swift account.
- **Container server**: Refers to the user-defined storage area within a Swift account where it stores a list of objects.
- **Object server**: Manages an actual object within a container. Object storage defines where the actual data and its metadata are stored. Each object must belong to a container.
- **Proxy server**: Handles different types of incoming object API or HTTP requests, such as container creation, object uploads, and deletion.
- **Partition**: Manages the location of objects, containers, and account databases.
- **Zones**: Separate objects within physical zone isolation to prevent wider data loss during a zonal failure in a cluster.
- **Rings**: Defines a logical mapping of objects, accounts, and containers to a physical location in a Swift cluster. Swift uses a ring per storage construct. It also uses `swift-ring-builder` to create an inventory of the cluster and create partitions.

As shown in the following figure, each partition will have three replicas that reflect three copies of the same object, container, or account to maintain high availability of data:

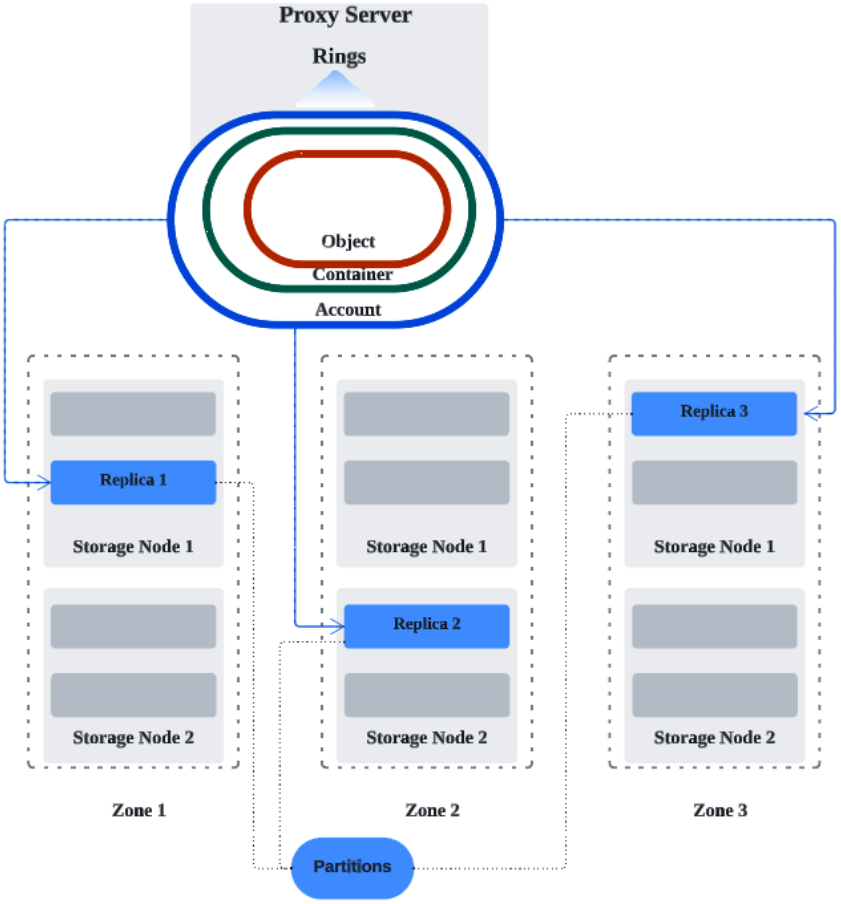


Figure 5.4 – Swift rings and replication in a three-zone layout

As well as the previous core components, Swift uses additional background daemons to manage objects' indexing, updates, and replications.

Building a Swift cluster

There are a variety of parameters that should be considered when preparing an object storage cluster before integrating it with an existing OpenStack environment. The focus should be on the hardware requirements to achieve a highly available, standalone, and expandable cluster. The following is a starting point for deploying a Swift cluster:

- **Object Storage capacity**: 100 TB
- **Number of hard drive slots per chassis**: 50

- **Hard drive storage capacity**: 3 TB
- **Recommended cluster replicas**: 3
- **Supported filesystem**: XFS

As we intend to keep three replicas, as recommended, the total storage capacity should be three times the recommended size, giving us `3 * 100 = 300 TB`.

The usage of XFS as a filesystem requires a metadata overhead in storage with a factor of 1.0526. The rounded required storage space, in this case, will be calculated as follows:

```
300 * 1.0526 = 316 TB
```

Using the total space required will help you find the number of hard drives required, as follows:

```
316/3 = 106 drives
```

The number of storage nodes will be determined as follows:

```
106 / 50 = 3 nodes
```

Our initial cluster will be composed of three storage nodes that will host containers, accounts, and objects. To manage the Swift requests, in our initial draft, we have assigned the `swift-proxy` server to run on the controller node. One of the caveats of keeping the `swift-proxy` role on the controller node is the performance risk that could overload the cloud controller resources. As a cloud operator, it is recommended to monitor the object API requests reaching the `swift-proxy` service and start to move it to dedicated hardware when the storage cluster continues to expand, in terms of size and usage.

Another critical consideration when dealing with an extended Swift storage system is the networking layout. In our initial design draft, a network interface is dedicated to handling storage. As per the Swift architecture, more interfaces can be assigned for different networking purposes. These are listed here:

- **Storage interface**: A proxy server that interfaces the storage nodes running the object, container, and account servers. The associated configuration in `kolla-ansible` is defined in the `globals.yml` file as `swift_storage_interface`.
- **Replication interface**: Optionally, an additional interface in each storage node can be dedicated to handling Swift storage replication between the storage nodes. The associated configuration in `kolla-ansible` is defined in the `globals.yml` file as `swift_replication_interface`. This option is not considered in the current setup and will use the storage interface to handle the replication traffic.
- **Internal API interface**: This can be defined to allow users access to the proxy server internally. The same interface can be used by HAProxy to establish load balancing between two or more swift-proxy services for high availability. The associated configuration in `kolla-ansible` is defined in the `globals.yml` file as `api_interface`. This option is not considered in the current setup.

- **External API interface**: Optionally, Swift can be accessed externally when workloads require public API access. The associated configuration in `kolla-ansible` is defined in the `globals.yml` file as `kolla_external_vip_interface`. This option is not considered in the current setup.

Swift can easily expand exponentially due to its common use cases; thus, cloud operators should expect possible growth of the underlying infrastructure. A common pain point is to get locked in a limited hardware configuration and not be prepared for it beforehand. Having additional network interfaces will prove a big advantage when operators realize there's a degradation in network performance. Separating the traffic types in a dedicated interface will avoid the burden of operational overhead and firefighting.

Running Swift

Before deploying the Swift cluster using `kolla-ansible`, we will need to prepare the storage node filesystem. The following shell script will create partitions on each available disk – `sdc`, `sdd`, and `sde` – that should run in each storage node:

```
for i in sdc sdd sde; do
     parted /dev/${i} -s -- mklabel gpt mkpart KOLLA_SWIFT_DATA 1 -1
done
```

For each created partition, create an XFS file system using the following shell script:

```
loop=0
for i in sdc sdd sde; do
     sudo mkfs.xfs -f -L i${loop} /dev/${i}1
    (( loop++ ))
done
```

> **Important note**
>
> The filesystem label is referred to as `KOLLA_SWIFT_DATA`, which should be visible in the storage node. The name of the device should match the configuration settings in the `globals.yml` file, defined by the `swift_devices_name` setting.

Next, generate the rings to prepare mapping objects in the Swift cluster. At the time of writing, the Object Storage playbook does not automate the ring generation process. Start by generating the object ring, and then run the following script from the deployer node. The first part of the script defines the cluster storage IP addresses, the Swift Kolla image, and a new directory for a custom Swift Kolla configuration path, respectively:

```
$ STORAGE_NODES=(10.30.0.81 10.30.0.82 10.30.0.83)
$ KOLLA_SWIFT_BASE_IMAGE="kolla/ubuntu-source-swift-base:8.0.5"
$ sudo mkdir -p /etc/kolla/config/swift
```

Swift uses the `swift-ring-builder` tool to generate an object ring. The generic format of the ring builder command is as follows:

```
swift-ring-builder <builder_file> create <part_power> <replicas> <min_
part_hours>
```

Here, we have the following:

- `<builder_file>`: This can be one of `account.builder`, `container.builder`, or `object.builder`.
- `<part_power>`: The number of partitions is approximated to the closest power of 2 to get the part power of the cluster. For example, for 50 hard disks, the recommended part power is 11, which gives us a value of 2,048 on average for all partitions. It is recommended that the approximation is rounded up.
- `<replicas>`: A choice of three replicas will be the recommended value.
- `<min_part_hours>`: This determines the time in an hour during which only one replica of a partition can be moved.

Rackspace Lab provides an online easy tool to calculate the swift ring. It can be found at `https://rackerlabs.github.io/swift-ppc/`.

The next part of the script will build the ring for the objects and iterate through each storage node by adding available devices to the ring on port `6000`:

```
docker run \
  --rm \
  -v /etc/kolla/config/swift/:/etc/kolla/config/swift/ \
  $KOLLA_SWIFT_BASE_IMAGE \
  swift-ring-builder \
    /etc/kolla/config/swift/object.builder create 11 3 1

for node in ${STORAGE_NODES[@]}; do
    for i in {0..2}; do
      docker run \
        --rm \
        -v /etc/kolla/config/swift/:/etc/kolla/config/swift/ \
        $KOLLA_SWIFT_BASE_IMAGE \
        swift-ring-builder \
          /etc/kolla/config/swift/object.builder add \
          r1z1-${node}:6000/d${i} 1;
    done
done
```

The following section of the script will generate a ring file for the account and add available devices to the ring for each storage node on port `6001`:

```
docker run \
  --rm \
  -v /etc/kolla/config/swift/:/etc/kolla/config/swift/ \
  $KOLLA_SWIFT_BASE_IMAGE \
  swift-ring-builder \
    /etc/kolla/config/swift/account.builder create 11 3 1

for node in ${STORAGE_NODES[@]}; do
    for i in {0..2}; do
      docker run \
        --rm \
        -v /etc/kolla/config/swift/:/etc/kolla/config/swift/ \
        $KOLLA_SWIFT_BASE_IMAGE \
        swift-ring-builder \
          /etc/kolla/config/swift/account.builder add \
          r1z1-${node}:6001/d${i} 1;
    done
done
```

The next part of the script generates a ring for the container and adds available devices to the ring for each storage node on port `6002`:

```
docker run \
  --rm \
  -v /etc/kolla/config/swift/:/etc/kolla/config/swift/ \
  $KOLLA_SWIFT_BASE_IMAGE \
  swift-ring-builder \
    /etc/kolla/config/swift/container.builder create 11 3 1

for node in ${STORAGE_NODES[@]}; do
    for i in {0..2}; do
      docker run \
        --rm \
        -v /etc/kolla/config/swift/:/etc/kolla/config/swift/ \
        $KOLLA_SWIFT_BASE_IMAGE \
        swift-ring-builder \
          /etc/kolla/config/swift/container.builder add \
          r1z1-${node}:6002/d${i} 1;
    done
done
```

Optionally, to distribute the partitions within the available drives in the ring, add the last section in the script by firing the rebalance option, using the same `swift-ring-builder rebalance` command-line tool:

```
for ring in object account container; do
  docker run \
    --rm \
    -v /etc/kolla/config/swift/:/etc/kolla/config/swift/ \
    $KOLLA_SWIFT_BASE_IMAGE \
    swift-ring-builder \
      /etc/kolla/config/swift/${ring}.builder rebalance;
done
```

This will iterate through each generated ring file and ensure each ring is balanced upon each partition move. After running the ring preparation script, make sure no errors are generated and that all three ring files are located under the `/etc/kolla/config/swift/` directory. The next step is to make sure the object storage hostnames are added to the inventory file, located under `/ansible/inventory`, as follows:

```
…
[storage]
storage03.os.packtpub
storage04.os.packtpub
storage05.os.packtpub
[swift-account-server:children]
storage
[swift-container-server:children]
storage
[swift-object-server:children]
storage
```

We will keep the `swift-proxy` role on the cloud controller node:

```
…
[swift:children]
control
[swift-proxy-server:children]
swift
```

Edit the `globals.yml` file by enabling the Swift service, as follows:

```
enable_swift : "yes"
```

If the storage partition label uses a different name than the default `KOLLA_SWIFT_DATA` defined in the disk partition preparation previously, make sure you adjust it accordingly by setting the following configuration line in the `globals.yml` file:

```
swift_devices_name: "KOLLA_SWIFT_DATA"
```

Triggering the pipeline job should run each target node in the Kolla Swift containers. Create a container in one of the cluster nodes, as follows:

```
$ openstack container create pp_container
```

Create a simple object in the created container:

```
$ openstack object create pp_container obj1.txt obj2.txt
```

You can show the container details, including the number of objects and the associated account:

```
$ openstack container show mycontainer

+--------------+----------------------------------------+
| Field        | Value                                  |
+--------------+----------------------------------------+
| account      | AUTH_4c32445a3d2e431f3ab3347e522556a9 |
| bytes_used   | 6778                                   |
| container    | pp_container                           |
| object_count | 2                                      |
+--------------+----------------------------------------+
```

As demonstrated in this section, the Swift architecture comes with a simple API that makes it easy to store unstructured data in redundant clusters with less operational burden. Swift is also often used for backup, archiving, and disaster recovery purposes. The last facet of storage options in OpenStack is the file share storage code known as Manila, which will be covered in the next section.

Exploring file share services – Manila

The next storage service in OpenStack is the file share code named Manila. The service offers simultaneous access to a storage-based file share for a variety of clients and storage backends. Multiple sharing protocols are supported by Manila, at least until the Bobcat release, including NFS, GlusterFS, CephFS, CIFS, HDFS, and, most recently, MapRFS. Note that the storage backends use drivers that must support one of the previously listed protocols.

Manila under the hood

As illustrated in the following figure, the core components that orchestrate file shares and manage their life cycle in Manila are as follows:

- **Manila share server**: A unit of storage that hosts the shares
- **API server**: Exposes a REST API interface for client request handling
- **Scheduler**: Selects the best share server to suit a file share request
- **Data service**: Handles data backups, recovery, and migration

Manila interacts with a few OpenStack core services, including the following:

- **Nova**: To create instances that run the share servers
- **Neutron**: To provide access to the file shares through the tenant instance network
- **Cinder**: To create file shares in a block storage pool as volumes:

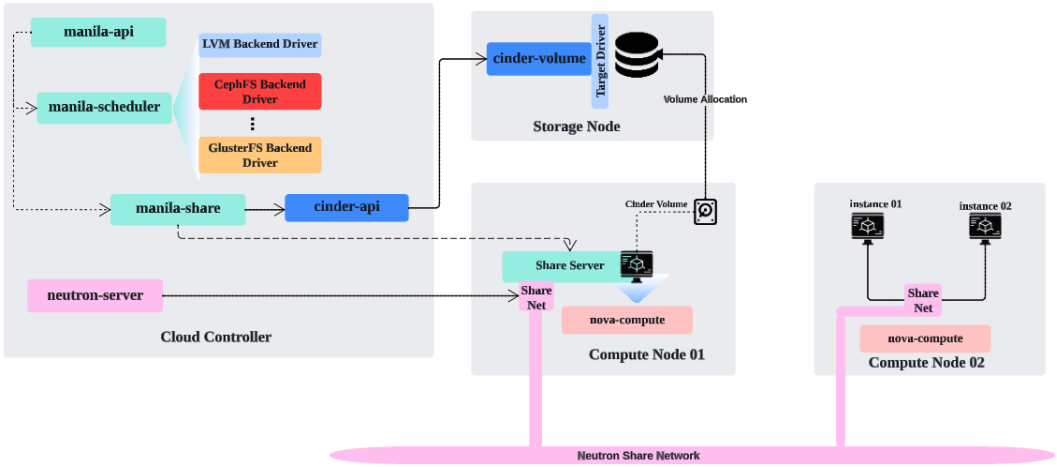


Figure 5.5 – File share service core architecture

Important note

At the time of writing, Manila supports more than 25 share drivers' backends, including CephFS, GlusterFS, LVM, and other vendors, such as EMC and Hitachi NAS. A fully updated list can be found at `https://docs.openstack.org/manila/latest/configuration/shared-file-systems/drivers.html`.

The additional extended list of new features and drivers in Manila allows it to use multiple storage backends and handle them simultaneously. Similar to the Block Storage multi-backend setup, the latest versions of Manila introduced the filtering and weighing mechanisms to select a shared backend, based on specific properties. By default, `manila-scheduler` uses `DriverFilter` for filtering and `GoodnessWeigher` as a weigher mechanism.

> **Important note**
>
> Filters and function weighers should be configured in the `manila.conf` file, similar to Cinder's filtering function syntax. The filter function defines capability checks at both the share and host levels. The available share types of properties can be listed by running `manila extra-specs-list` on the command line. A full list of the latest properties supported by the filter function, up until the Antelope and Bobcat releases, is available at `https://docs.openstack.org/manila/latest/contributor/driver_filter_goodness_weigher.html`.

For the next deployment, the default scheduling configuration will be used, which filters the best backend based on the free available capacity.

Running Manila

Deploying the Manila service in our existing OpenStack environment is straightforward. In the following setup, we will use a generic backend that leverages Cinder to create a share as a volume. The attached volume to the instance can live within a Cinder pool storage backend that was configured in the *Expanding storage with multiple backends* section for LVM or Ceph, respectively.

Enabling the Cinder service in the `kolla-ansible` code is a requirement to run Manila with a generic storage backend. In the `globals.yml` file, enable the Manila service, as follows:

```
enable_manila: "yes"
```

The `globals.yml` file exposes another backend for Manila that includes NFS in Ceph and HNAS. For the generic backend, bear in mind that NFS and CIFS shares will be used. Proceed by enabling the generic share:

```
enable_manila_backend_generic: "yes"
```

> **Important note**
>
> At the time of writing, enabling the Manila service will automatically add the Horizon panel for Manila to manage file shares through the dashboard. If the Manila service is not visible in the dashboard after deployment, make sure the `enable_horizon_manila` setting is set to `true`.

More Manila settings can be configured in the custom Manila configuration file, which is not controlled directly via the `globals.yml` file. Create a new configuration file, named `/etc/kolla/config/manila-share.conf`, to customize the Manila share settings. One considerable setting is the size of the instance share required. You can create the instance flavor in Nova in advance or use the default ones, which can be listed using the following command line:

```
$ openstack flavor list
```

For example, we can instruct the file share instance to use the `m1.medium` flavor, referenced with an ID of `3` from the list:

```
[generic]
service_instance_flavor_id = 3
```

Other settings can be customized, such as the Manila shared backend name if you intend to have more than one backend. To reflect the name of the generic backend, add the following configuration setting:

```
[generic]
…
share_backend_name = GENERIC_SHARE
```

As it's a Nova instance, the file share server will need a Glance image. By default, the Manila playbook uses `manila-service-image` as a reference name. If you intend to change it, make sure it has the same name as the one configured in the file.

The next part of preparing the Manila deployment is to make sure the Manila services are assigned to the cluster roles. Set the hostname, and ensure the Manila services are added to the inventory file located under `/ansible/inventory/multi_packtpub_prod`, as follows:

```
…
[manila]
control
[manila-api:children]
manila
[manila-scheduler:children]
manila
[manila-data:children]
manila
```

As highlighted in *Chapter 3*, *OpenStack Control Plane – Shared Services*, the `manila-share` service uses the Neutron plugin to provide network access to the file share resources:

```
…
[manila-share:children]
network
```

Launch the pipeline job so that you can deploy the Manila service in cross-target nodes. Manila service containers should be running in the cloud controller and network node. Make sure that the Manila tasks complete without failure, as shown here:

```
TASK: [manila | Creating Manila database] ***********************************
skipping:

TASK: [manila | Reading json from variable] **********************************
skipping:

TASK: [manila | Creating Manila database user and setting permissions] ********
skipping:

TASK: [manila | Running Manila bootstrap container] *************************
skipping:

TASK: [manila | Starting manila-api container] *******************************
skipping:

TASK: [manila | Starting manila-scheduler container] *************************
skipping:

TASK: [manila | Starting manila-share container] ****************************
skipping:

PLAY RECAP *********************************************************************
localhost                  : ok=311  changed=123  unreachable=0    failed=0
```

Figure 5.6 – The kolla-ansible Manila deployment's service output

Verify the Manila services in the cloud controller node:

```
$ openstack share service list
```

The output of the previous command line shows that the `manila-scheduler` service has been enabled on the cloud controller host and that `manila-share` is running as part of the network node:

```
+------------------+----------------+------+---------+-------+----------------------------+-----------------+
|      Binary      |    Host        | Zone |  Status | State |         Updated_at         | Disabled Reason |
+------------------+----------------+------+---------+-------+----------------------------+-----------------+
| manila-scheduler | controller     | nova | enabled |   up  | 2023-11-19T22:20:44.000000 |       None      |
| manila-share     | net01.os       | nova | enabled |   up  | 2023-11-19T22:20:47.000000 |       None      |
+------------------+----------------+------+---------+-------+----------------------------+-----------------+
```

Figure 5.7 – The OpenStack Manila services list

The file-sharing service has seen great improvements in the latest OpenStack releases, including Bobcat, such as the introduction of file-share control access. This way, file shares can be locked to protect them against accidental deletion.

Important note

Starting from the Antelope release, the Manila CLI is planned to be deprecated, and it is highly recommended to use the OpenStack CLI to interact with the Manila REST API.

Summary

This chapter has explored the most trending OpenStack storage services. The expansion of each storage service feature and capability, including Swift, Cinder, and Manila, is almost *unstoppable*. This is confirmed by the vast array of supported backend drivers, either with the open source world ones, such as Ceph, or with a specific storage vendor. The latest OpenStack releases, including Antelope and Bobcat, have also introduced more stable advanced features, mainly for Cinder and Manila, through the scheduling and filtering mechanisms. Bear in mind that understanding each storage use case and its respective architecture are the keys to comfortably running a storage solution in an OpenStack environment, based on your requirements. The challenge of picking the right option to satisfy specific requirements is not limited to storage topics. It also applies to the networking paradigm in OpenStack. Connecting the different elements of the compute and storage services will be discussed in the next chapter of our OpenStack journey.

6

OpenStack Networking – Connectivity and Managed Service Options

"It always seems impossible until it's done."

– Nelson Mandela

Neutron, the OpenStack networking service, has evolved throughout different OpenStack releases, offering more capabilities and features. Cloud architects can design their OpenStack networking stack with multiple topologies, based on their existing networking resources and requirements. Unlike the former Nova networking service, which was bound only to basic networking offerings, OpenStack tenants can expect more granular networking control and flexibility with Neutron. Cloud and network administrators can offer more advanced networking features to tenants, such as routing, firewalls, load balancing, and private networking.

In this chapter, we will walk through the rich updates that have been made to OpenStack networking by covering the following topics:

- The Neutron architecture and its core service interactions in OpenStack
- Neutron plugins and supported drivers
- Existing Neutron agents and required network types
- Designing and implementing an OpenStack virtual switching layout using **Open vSwitch** (**OVS**) and **Open Virtual Network** (**OVN**) mechanisms as a **Software-Defined Networking** (**SDN**) implementation

- Learning about and implementing routing, using virtual routers to interconnect tenant networks and provide external connectivity
- Integrating the load balancer as a service feature using the emerging Octavia project

Exploring Neutron's core components

OpenStack networking has evolved through different releases to build an advanced and functioning cloud networking stack. Like many other OpenStack services, Neutron is composed of several services that can be stretched across multiple hosts, as shown in the following diagram:

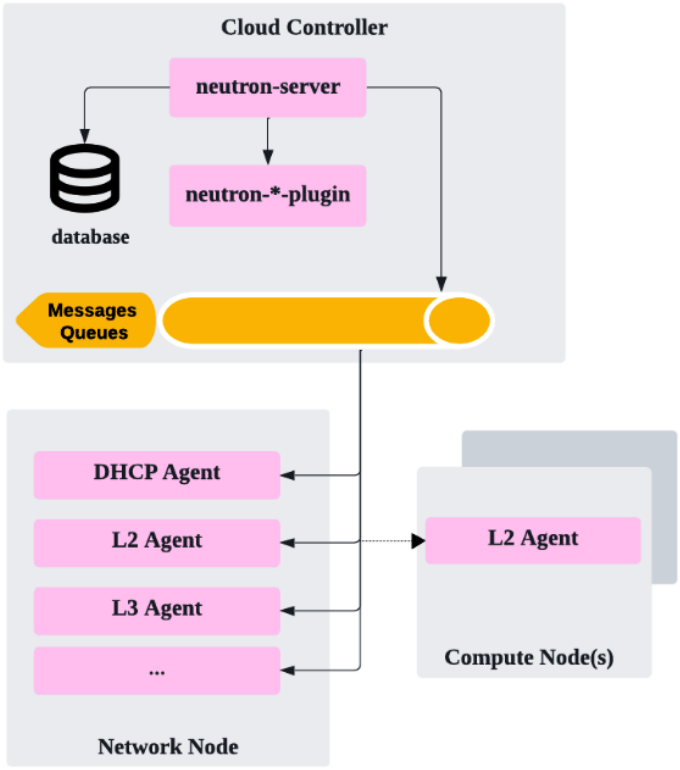


Figure 6.1 – The OpenStack Neutron core architecture

These components can be briefly broken down as follows:

- **Neutron server**: This acts as an API portal that receives API requests generated by services or end users and forwards them to the next process – in this case, the Neutron agents through the messaging queue service. The other part of the server interaction within an OpenStack ecosystem is access to the database to update the network objects for each API request.

- **Neutron agents**: Neutron's architecture relies heavily on different types of agents that will be installed in other hosts to handle different networking features. The plugin agent, denoted as `neutron-*-agent`, is the process that handles virtual switching capabilities in each compute node. This type can be described as a **layer 2** (**L2**) agent. A very common L2 agent is OVS, which provides L2 connectivity via the ML2 mechanism driver. The second type of Neutron agent is the **layer 3** (**L3**) agent, denoted as `neutron-l3-agent`, which is installed in the network node and deals with instances of network access for L3, such as NAT, firewall, and **virtual private network** (**VPN**) capabilities. The DHCP agent, denoted as `neutron-dhcp-agent`, manages instances of DHCP configuration for each tenant network (a `dnsmasq` configuration).

Depending on the nature of the network function request, the Neutron service will reach the agents deployed on the network and/or compute nodes. Cloud operators should list which network functions the infrastructure supports, which will decide the choice of plugins and agents that can be installed.

Demystifying Neutron agents

Neutron relies on agent services that interact with `neutron-server` through the queuing message service to implement virtual networking for L2 and L3 connectivity, DHCP, and routing services. Different agents will be deployed in the network and compute nodes:

- **L2 agent**: Deployed on network and compute nodes to implement L2 networking connectivity for instances and virtual networks, such as routers.
- **L3 agent**: Deployed on the network node and, optionally, on compute nodes to perform L3 routing between different types of networks, such as tenant and external networks. L3 can also implement more advanced networking capabilities, such as firewalls and VPNs.
- **DHCP agent**: Deployed on the network node to run the DHCP service for tenant networks.
- **Plugin agent**: Deployed on the network node to process data packets on virtual networks that depend on the implemented Neutron plugin.
- **Metadata agent**: Deployed on the network node. The metadata service provided by Nova enables the retrieval of instance information, such as hostname and IP addresses, upon a metadata HTTP request (`169.254.169.254`). The Neutron metadata agent facilitates the forwarding of metadata to the Nova metadata service through its internal proxy component with additional information, such as instance and tenant IDs.

The proper function of the deployed Neutron agent will vary, based on the types of the designed networks in the OpenStack environment. The next section will briefly cover the different categories of networks that should exist in our initial design draft.

Network categories

It is essential to go through the network types to enable traffic flow, based on their usage. The following glossary covers the network categories that are used mainly with the latest releases of OpenStack:

- **Provider networks**: Created and managed by cloud operators who define a set of network attributes, such as the network's type – for example, **VXLAN**, **Generic Routing Encapsulation** (**GRE**), or **flat**. Cloud operators provide and configure the underlying infrastructure, such as the physical network interface designated for the traffic flow.
- **Self-service networks**: Referred to as **tenant networks**, these are created by cloud users. Self-service networks are self-contained and fully isolated from other networks in a multi-tenancy environment managed by Neutron. Tenants are restricted to creating virtual networks with what was predefined by the cloud operators in terms of network types. For example, a cloud user cannot implement GRE networks if the cloud operator does not provide the option. Cloud users do not have access to the underlying Neutron configuration.
- **External provider networks**: This refers to a provider network for specific external network connectivity. Cloud operators configure an external routing device to access the internet.

The next section will cover the most interesting capabilities of Neutron by exploring the concept of plugins and their latest mode of usage.

The core of networking – Neutron plugins

Neutron supports the use of plugins and drivers, which use different software and hardware technologies provided by open source communities or vendor solutions. There are two types of Neutron plugins:

- **Core plugin**: Enables L2 connectivity functions and orchestration of network elements, including virtual networks, subnets, and ports
- **Service plugin**: Enables additional network capabilities, including routing, private networks, firewalls, and load balancing

> **Important note**
>
> The code for plugins is run on the Neutron server, which can be configured as part of the cloud controller node(s).

The following section will dive into the most adopted core plugins.

Core plugin – Modular Layer 2

The most widely adopted core plugin in Neutron is **Modular Layer 2** (**ML2**). The introduction of ML2 to the OpenStack networking service has increased the flexibility of designing network architectures.

The secret sauce of ML2 is that it supports the use of a variety of technologies from different vendors simultaneously. Thus, cloud architects and operators aren't limited to a specific set of features and can easily extend or change their network stacks.

This plugin has been developed due to the historical limitations prior to the **Havana** release, where operators had to stick to only one core plugin, referred to as a monolithic plugin-based component. The two most commonly used core plugins were **Linux Bridge** and **OVS**, which have been replaced in the latest releases of OpenStack by ML2. Replacing does not mean directly removing any of them but bundling them as **mechanism drivers** in the core ML2 plugin. ML2 is not limited to Linux Bridge and OVS and also supports other vendor drivers, such as VMware, Cisco, and OpenDayLight. An amazing filtered list can be found on the OpenStack community website: `https://www.openstack.org/marketplace/drivers/#project=neutron%20(networking)`.

An overview of the modular plugins and drivers supported in Neutron is illustrated here:

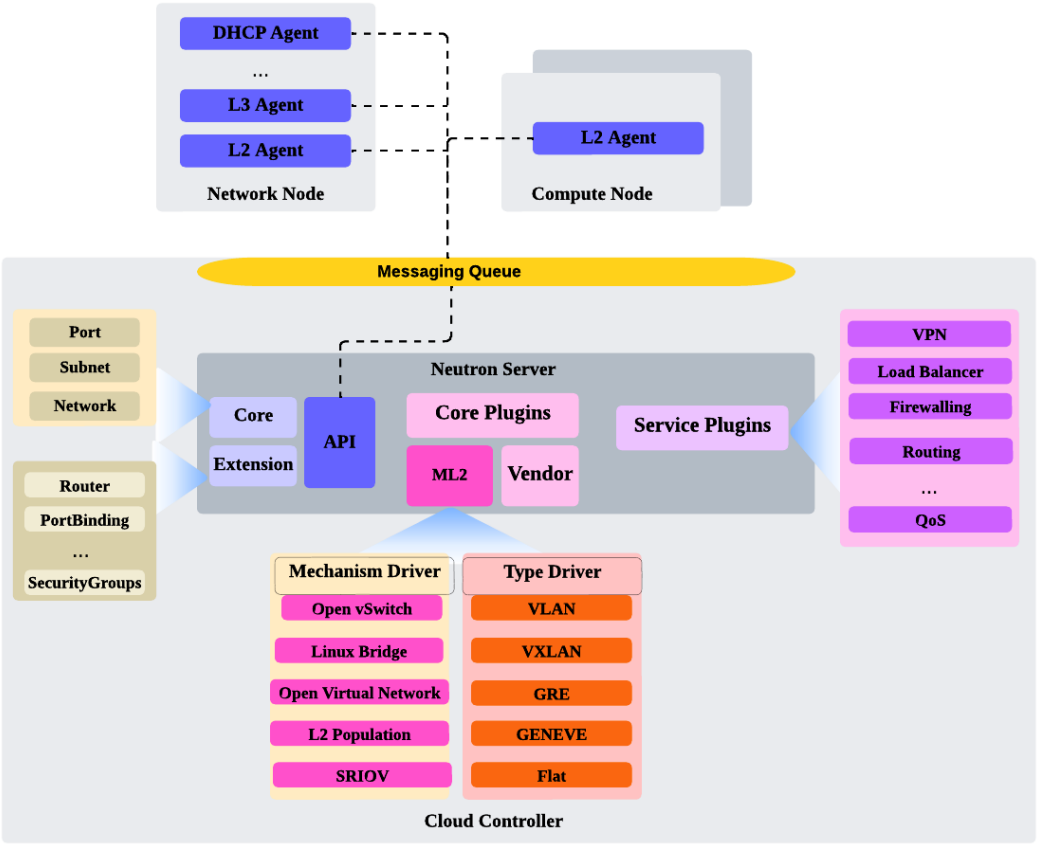


Figure 6.2 – An overview of the OpenStack Neutron core and service plugins architecture

As shown in the preceding diagram, Neutron relies on the mechanisms of plugins that provide a specific set of networking services. An API request will be forwarded by the Neutron server to the associated plugin configured in the controller node. The ML2 plugin combines two main elements into the same framework, as follows:

- **Type drivers**: These expose L2 functionalities to create more segmented networks, including the following:
  - **VLAN**: Segregates network traffic using **802.1Q** tagging. Virtual machines that belong to the same VLAN are part of the same broadcast L2 domain
  - **VXLAN**: Uses a **virtual network identifier** (**VNI**) to separate and differentiate traffic between different networks
  - **Flat**: VLAN tagging and network segregation are not supported where instances are connected in the same network
  - **GRE**: Encapsulates traffic using the GRE tunneling protocol
  - **GENEVE**: Resembles the VXLAN overlay technology but with more optimal encapsulation methods
  - **Local**: Connects instances only within the same network hosted in the same compute nodes and not with instances in different nodes
- **Mechanism drivers**: Implements the type driver technologies through both network software methods such as OVS, Linux Bridge, and OVN and hardware-based ones

An up-to-date ML2 driver support matrix list can be found at `https://docs.openstack.org/neutron/latest/admin/config-ml2.html`. Each mechanism driver supports a set of network types (type drivers). For example, OVS supports most network types, including VLAN, VXLAN, GRE, flat, and local. If a need has been raised to support the GENEVE type driver while in production, the new mechanism driver can be integrated by just installing the driver and reconfiguring the Neutron driver list.

Building virtual switching

In this section, two mechanism drivers will be discussed to establish connectivity between virtual network ports and physical networks. OVS is one of the most featured mechanism drivers, with advanced networking capabilities such as **OpenFlow**. The second driver we will cover is **OVN**, an SDN implementation that enables programming networks by controlling the flow of packets. Linux Bridge and **L2 Population** are also well-tested and mature mechanisms. With ML2 being an agnostic plugin for the supported drivers, introducing a new mechanism driver will not introduce a lot of complexity in an existing running OpenStack environment.

Opening vSwitch in OpenStack

Most OpenStack networking implementations would require a minimum of overlay networking technologies that provide tunnel-based virtual networking, with an immense level of network segmentation, such as VXLAN and GRE. Tunnel-based networks can provide up to 16 million networks. OVS supports most used drivers in large and complex OpenStack networking environments, such as VLAN, VXLAN, GRE, and flat. As a cloud operator, it is crucial to have a basic understanding of how OVS operates when installed in an OpenStack environment. OVS comes with a complete core architecture, with different components that operate as software switches in a host kernel space. The OVS implementation comes with different services when installed in an OpenStack environment, as follows:

- `openvswitch`: A kernel module that handles the data plane that processes the network packets.
- `ovs-switchd`: A Linux process that runs on a physical host to control and manage virtual switches.
- `ovsdb-server`: A local database to store the virtual switches running locally.
- `neutron-openvswitch-agent`: Configured in compute nodes that use the OVS mechanism driver.
- `neutron-server`: Handles API requests, with the ML2 plugin loading the OVS mechanism driver. The neutron-server process passes the network request to the OVS driver via an RPC cast message to `neutron-openvswitch-agent`. The latter configures the OVS switch on the compute node and sets the required resources of the local instance.

Once OVS is implemented, a set of virtual network devices is installed, which a cloud operator should keep in mind during deployment or troubleshooting tasks. An Ethernet frame would travel from an instance through the following interfaces:

- `tapXXXX`: A tap interface where `XXXX` is the assigned tap interface ID.
- `br-int`: A bridge interface, known as the integration bridge, where it consolidates all the virtual devices, such as virtual machines, routers, and firewalls.
- `int-br-ethX`: A virtual patch port connecting different interfaces to the OVS integration bridge interface (`br-int`).
- `phy-br-ethX`: A virtual patch port connecting different interfaces to the OVS provider bridge (`br-ethX`).
- `br-ethX`: A bridge interface where *X* is the assigned bridge interface ID connecting to the physical network. It is also known as the provider bridge and connects to the `br-int` integration bridge through a virtual patch cable, provided by patch ports (`int-br-ethX` and `phy-br-ethX` patch ports for `int-br` and `br-ethX`, respectively).
- `qbrXXXX`: A Linux bridge interface where `XXXX` is the assigned bridge interface ID dedicated to IP tables only.

- `br-tun`: A bridge tunnel interface to handle packet encapsulation and de-encapsulation.
- `br-ex`: A bridge interface providing connectivity to external networks.

Each spawned instance is connected via its tap interface, denoted as `tapXXXX`, created in the hypervisor host. The associated Linux bridge, denoted as `qbrXXXX`, is connected to the OVS `br-int` integration bridge, where the traffic will be routed based on the programmed OpenFlow rules and then forwarded through the virtual switch.

Connections between the integration and provider bridges are handled by patch cables. Packets exit the physical network via the OVS provider bridge (`br-ethX`), connected to the physical network interface of the host. When configuring OVS in our existing OpenStack environment, each node will run its own integration and provider bridges. Connectivity between the OpenStack nodes, including the cloud controller, network, and compute nodes, is established via the physical network, as illustrated in the following diagram:

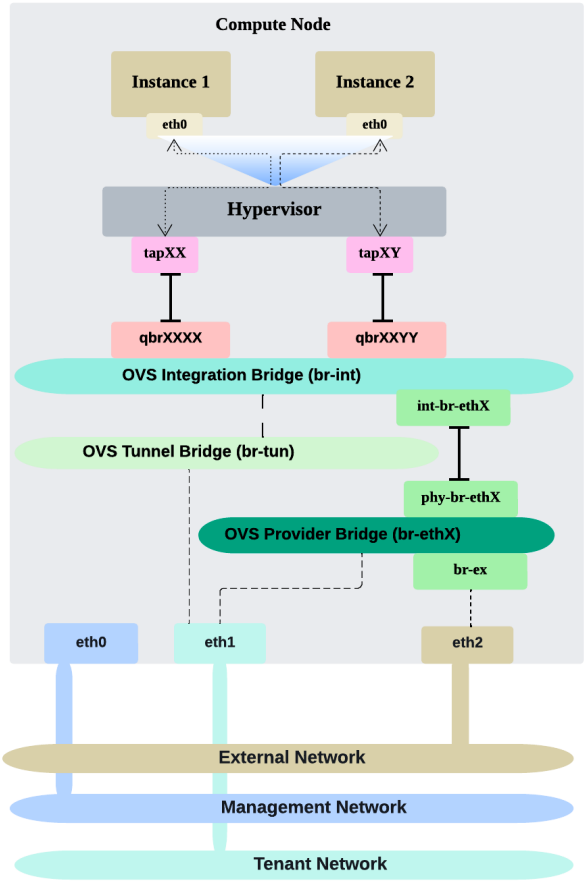


Figure 6.3 – Traffic flow through different interfaces using OVS

Now that we understand the basic core components of OVS within an OpenStack setup, we can move on to configuring our infrastructure code to deploy OVS as the main mechanism driver in Neutron.

Deploying with OVS

The OVS mechanism driver implements L2 isolation, which offers VXLAN-, GRE-, and VLAN-based networks. The next configuration will implement the ML2 plugin to use VXLAN-based networks. This will allow instances to connect via VLAN segmentation and VXLAN tunneling. Virtual networks for tenants will not be exposed outside of the compute or network nodes. `kolla-ansible` comes with the ML2 plugin enabled by default. A few prerequisites must be fulfilled before deploying an `OVS` configuration. To reach the external networks via routers and provider networks, adjust the `neutron_external_interface` setting in the `/etc/kolla/globals.yml` file to the network interface dedicated to this matter. The `eth2` interface has been assigned an interface to connect with the external world, as designed in *Chapter 3*, *OpenStack Control Plane – Shared Services*. You can adjust the settings as follows:

```
...
neutron_external_interface: "eth2"
...
```

Important note

It is important to consider the usage of a physical interface with the OVS provider bridge (`br-ethX`) if you are planning to use more than one provider bridge. In this case, `neutron_external_interface` can be assigned a comma-separated list – for example, `neutron_external_interface: "eth2,eth3"`.

The other check is to verify whether `kolla-ansible` is configured with an OVS driver mechanism that comes with the ML2 plugin. Check and adjust the following configuration setting in the `globals.yml` file:

```
...
neutron_plugin_agent: "openvswitch"
...
```

The next check is to verify that our Neutron services are assigned to the respective OpenStack nodes in the `/ansible/inventory/multi_packtpub_prod` file. As demonstrated in *Chapter 3*, *OpenStack Control Plane – Shared Services*, `neutron-server` will run as part of the controller node, as follows:

```
...
[control]
cc01.os.packtpub
...
```

```
[neutron-server:children]
control
...
```

Neutron agents, including the DHCP, L3, and metadata agents, will run in a dedicated network node, as follows:

```
...
[network]
net01.os.packtpub
[neutron:children]
network
[neutron-dhcp-agent:children]
neutron
[neutron-l3-agent:children]
neutron
[neutron-metadata-agent:children]
neutron
...
```

OVS will be installed in the network and compute nodes.

> **Important note**
>
> As shown in *Chapter 5*, *OpenStack Storage – Block, Object, and File Shares*, the `manila-share` service provides access to the file share by using the Neutron plugin.

The OVS setup in the inventory file is listed as follows:

```
...
[openvswitch:children]
network
compute
manila-share
...
```

Launch the pipeline job to install OVS across the target nodes. Note that VXLAN was not explicitly configured in the `kolla-ansible` code. By default, the Neutron tenant network uses the VXLAN type. The default VNI range is defined from 1 to 1,000 IDs, reserved for the tenant networks when they are created. This setting is denoted with the `vni_ranges` line in the `ml2_config.ini` file in the controller node. The ML2 configuration can be extended and we can employ a configuration override by creating an `/etc/kolla/config/neutron/ml2_config.ini` file.

The following command line lists the different network agents, including the OVS ones, from the cloud controller node:

```
$ openstack network agent list
```

Here is the output:



id	agent_type	host	alive	admin_state_up	binary
3ab3a7e5-bbf9-4ac8-8b9f-2044e021cf23	Metadata agent	net01.os	:-)	True	neutron-metadata-agent
13c25ea9-e558-4b69-af27-fb1ea8824e5a	L3 agent	net01.os	:-)	True	neutron-l3-agent
3a71878e-ac49-4a60-77d3-af3793705460	Open vSwitch agent	net01.os	:-)	True	neutron-openvswitch-agent
4e9df1b2-4fb7-4d01-8758-e5155448b0c8	Open vSwitch agent	cn02.os	:-)	True	neutron-openvswitch-agent
695c19de-4362-3a4e-88b1-aa0092e62169	DHCP agent	net01.os	:-)	True	neutron-dhcp-agent
e493a3a3-000c-4172-8a79-72e8f0d357e2	Open vSwitch agent	cn01.os	:-)	True	neutron-openvswitch-agent

Figure 6.4 – The neutron agent list output

As shown here, the OVS agents should be up and running in the respective network and compute nodes. In the following section, we will trace the network flow with the OVS mechanism.

Traffic flow with OVS

The setup of the OVS driver will dictate the travel hops of an Ethernet frame, from an instance all the way to the physical network. Numerous interfaces will be used, including the virtual switch ports and integration bridge. The deployment of the `openvswitch` playbook in the compute node should create the OVS logical bridges on the host, which can be checked by running the following command line:

```
$ ovs-vsctl list-br
```

The output will be as shown:

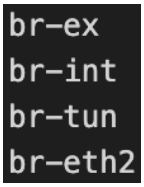


Figure 6.5 – The OVS bridges list

Instances spawned in the compute node are connected via the same local VLAN. With the VXLAN layout, instance ports are attached to the integration bridge, which will be assigned a local VLAN ID. Traffic initiated from an instance will be forwarded by the integration bridge (`br-int`) to the tunnel bridge (`br-tun`). The overall layout of the connecting bridges and their respective ports can be seen by running the following command line:

```
$ ovs-vsctl show
```

Here is the output:

```
1a12accd-54ea-2217-65ea-de3da45a12a8
    Bridge br-tun
        fail_mode: secure
        Port "vxlan-476984a0"
            Interface "vxlan-476984a0"
                type: vxlan
                options: {key="6000", remote_ip="10.10.0.165"}
    Bridge br-int
        fail_mode: secure
        Port br-int
            Interface br-int
                type: internal
        Port "qr-476984a0-53"
            tag: 1
            Interface "qr-476984a0-53"
                type: internal
        Port "tapa3281257-92"
            tag: 1
            Interface "tapa3281257-92"
                type: internal
```

Figure 6.6 – An OVS interface listing

As depicted in the virtual switch interfaces, the integration bridge (`br-int`) is tagged with `VLAN 1`. When traffic reaches instances in different nodes, the packets will travel through the tunnel bridge (`br-tun`) encapsulated in VXLAN packets. Under the hood, the local VLAN ID of `1` will be swapped with the VXLAN tunnel ID, as shown with `vxlan-476984a0`. Otherwise, if an instance connects to another one residing in the same compute node, the packets will travel through the integration bridge (`br-int`) locally to the destination port, denoted with the `qrXXXX` and `tapYYYY` port interfaces.

The following section covers a more complex mechanism driver, OVN, and demonstrates its deployment in our OpenStack environment.

OVN in OpenStack

The other major and preferred network virtualization in an OpenStack environment is OVN. OVN is considered a mature implementation of the SDN philosophy that provides greater flexibility for network flow programming. Operators can define a set of forwarding rules centrally to define how the flow of packets is controlled. OVN uses an abstraction software layer available in a network controller to program switches, with conditions and rules applied to the packet flow. The most distinctive functionality of OVN compared to OVS is its richer capabilities and advanced features, such as traffic routing and access control which are programmable. Unlike the OVS approach, OVN decouples the control function of network devices from the actual packet forwarding function through flow rules. OVN supports most network types, including VLAN, VXLAN (version 20.09 and later), flat, and Geneve.

As depicted in the following diagram, OVN is built on top of OVS, which leverages its switching capabilities and adds an abstraction layer, managed by controllers that are stored in a set of **OVS databases** (**OVSDBs**):

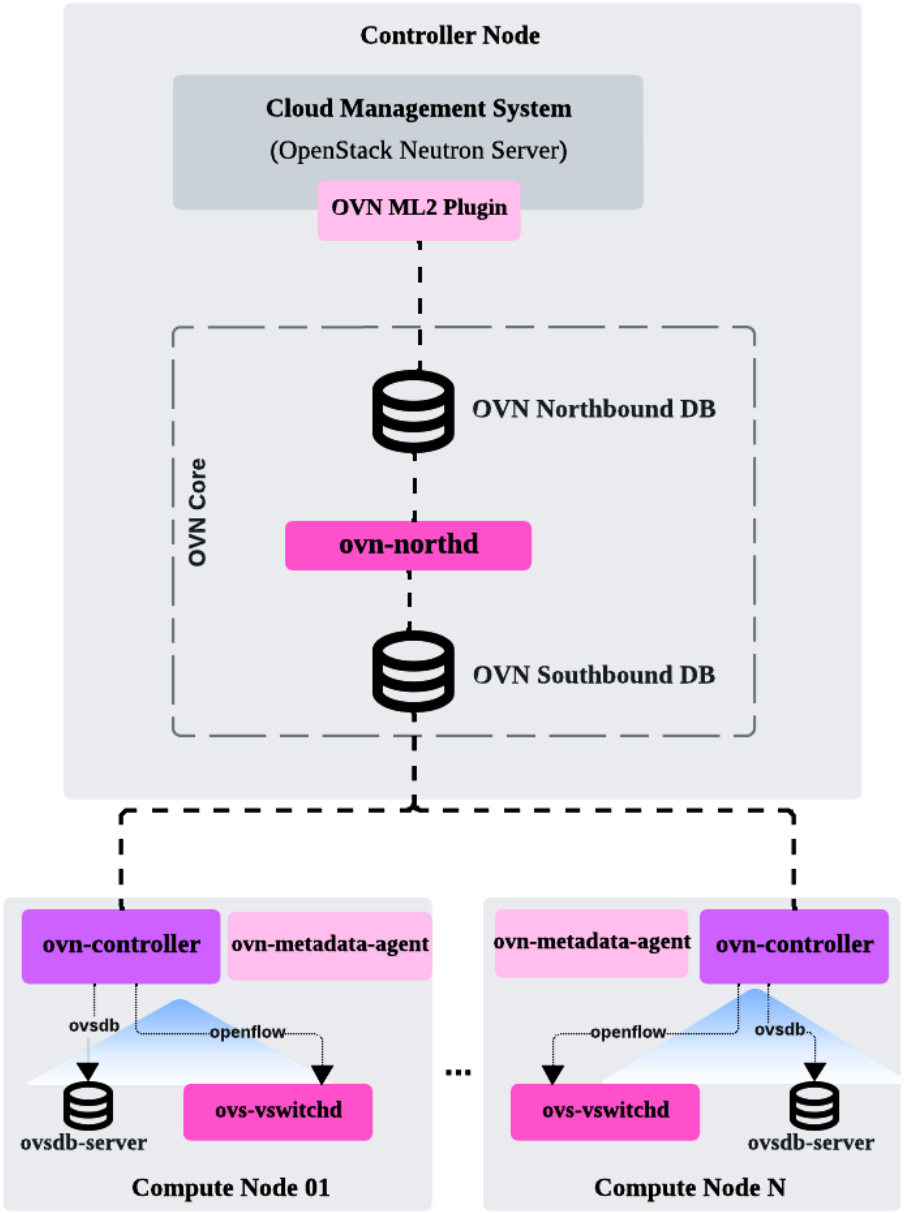


Figure 6.7 – OVN integration architecture

The main constructs of the OVN architecture integration with an OpenStack environment are explained here:

- **Southbound database**: Referenced as `ovsdb-server` (`ovnsb.db`), this stores data of the logical and physical network flows. The database hosts all binding information between the logical and physical networks, such as the port and logical networking bindings associated with the `Port_Binding` and `Logical_Flow` tables, respectively.
- **Northbound database**: Referenced as `ovsdb-server` (`ovnnb.db`), this stores data at a high level of the virtual networks that represent a **cloud management system** (**CMS**) – in this case, an OpenStack environment. The database reflects the networking resources of the CMS in the table datasets, such as the routers and switch ports associated with the `Logical_Router` and `Logical_Switch_Port` tables, respectively.
- **OVN controller**: Referenced as `ovn-controller`, this runs on hypervisor nodes and connects to the southbound database. It acts as the OVS controller by defining the OpenFlow rules on each OVS switch instance, converting logical flows into physical flows, and pushing configurations to the local OVSDB.
- **OVN northbound service**: Referenced as `ovn-northd`, this runs as part of the control plane on the cloud controller. The `ovn-northd` daemon connects the northbound to the southbound database by translating the logical configuration from the `ovn-nb` data to logical data path flows. The converted data will be populated in the `ovn-sb` database and made ready for use by the `ovn-controller` instance.
- **OVS data plane service**: Referenced as `ovs-vswitchd`, this runs as part of the data plane on the hypervisor nodes. `ovs-vswitchd` applies the packet-forwarding rules provided by `ovn-controller`.
- **Local OVSDB**: Referenced as `ovsdb-server`, this runs locally in each hypervisor node. The OVN reference architecture takes advantage of the database scaling benefit by using a consistent snapshot of the database, to recover from possible connectivity interruptions.
- **Metadata agent**: Referenced as `ovn-metadata-agent`, this runs in each hypervisor node. The OVN metadata agent, such as the Neutron metadata agent, is used to proxy metadata API requests on each compute node. This way, each OVS switch instance running in each hypervisor node will route the requests to the local metadata agent and then query the Nova metadata service to run instances.

In the following section, we will use `kolla-ansible` to integrate the OVN project in an OpenStack environment.

Deploying with OVN

OVN extends the OVS implementation to provide network services to instances. The ML2 mechanism driver facilitates integration with OpenStack as a CMS. The `kolla-ansible` playbooks include most of the OVN configuration and required packages to install. Similar to the OVS configuration, adjust `/etc/kolla/globals.yml` to use the designated interface to reach external networks, by configuring the `neutron_external_interface` setting, as follows:

```
...
neutron_external_interface: "eth2"
...
```

Configure the ML2 mechanism driver to use OVN, as follows:

```
...
neutron_plugin_agent: "ovn"
...
```

OVN comes with built-in support for L3 networking functions that enable instances to connect to external networks. There are two ways to provide such connectivity in OVN – through a centralized layout that does not use a distributed floating IP design or via **distributed virtual routing** (**DVR**), which requires a floating IP configuration. For two instances to reach external networks or each other using L3, traffic must flow through the network node. With the DVR method, traffic is more optimized by deploying an L3 agent in each compute node, allowing `north-south` (going to and from external networks) traffic with a floating IP to be routed from and to the compute node, with an extra hop to reach the network node. Similarly, `east-west` (traffic flow between instances) traffic is routed directly to the compute nodes. The latter option is recommended for more optimal performance. The distributed floating IP option can be enabled in the `globals.yml` file, as follows:

```
...
 neutron_ovn_distributed_fip: "yes"
...
```

Enable the Neutron OVN agent to provide extensible capabilities for network monitoring and **Quality of Service** (**QoS**) by configuring the following setting:

```
...
 neutron_enable_ovn_agent: "yes"
...
```

Next, make sure to adjust the `/ansible/inventory/multi_packtpub_prod` inventory file to install the OVN services in the assigned nodes. Similar to the OVS configuration, make sure to have the Neutron server and respective agents running in the controller and network nodes, respectively, as follows:

```
...
[control]
cc01.os.packtpub
...
[neutron-server:children]
control
[network]
net01.os.packtpub
[neutron:children]
network
[neutron-dhcp-agent:children]
neutron
[neutron-l3-agent:children]
neutron
[neutron-metadata-agent:children]
neutron
...
```

As discussed in the *OVN in OpenStack* section, the OVS northbound and southbound databases, in addition to the OVN `northd` daemon, will run on the controller node, as follows:

```
...
[ovn-database:children]
control
[ovn-northd:children]
ovn-database
[ovn-nb-db:children]
ovn-database
[ovn-sb-db:children]
ovn-database
```

The OVN controller instances and OVN metadata agents will run on the compute nodes, as follows:

```
...
[neutron-ovn-agent:children]
compute
[ovn-controller:children]
ovn-controller-compute
[ovn-controller-compute:children]
compute
```

```
[neutron-ovn-metadata-agent:children]
compute
...
```

Important note

Since Antelope and later releases, a dedicated OVN agent has been created to implement the lacking `ovn-controller` function.

Launching the pipeline job should deploy the OVN services and create an ML2 file, `/etc/neutron/plugins/ml2/ml2_conf.ini`, with the following default configurations:

```
...
[ml2]
tenant_network_types = geneve
type_drivers = local,flat,vlan,geneve
mechanism_drivers = ovn
[ml2_type_geneve]
max_header_size = 38
vni_ranges = 1001:2000
...
```

The default network tenant type is configured with GENEVE as the tunneling protocol. The main difference between GENEVE and VXLAN (configured for OVS in a previous section) is how the networking metadata is encapsulated and encoded between both protocols. VXLAN encodes the VNI only in the encapsulation header. Meanwhile, the GENEVE protocol uses extensible TLV to carry more information about the packet in the encapsulated header (it has a header length of 38 bytes, while the VXLAN frame can carry 8 bytes), such as the `ingress` and `egress` ports. GENEVE offers better capabilities, such as transport security, service chaining, and in-band telemetry, compared to VXLAN.

The other resulting default configuration is the default northbound and southbound database addresses. Each OVN controller running in each compute node should be able to reach both databases. Such configuration can be found in the same file, `ml2_conf.ini`, by checking the following settings:

```
...
[ovn]
ovn_nb_connection = tcp:10.0.0.24:6642
ovn_sb_connection = tcp:10.0.0.24:6642
ovn_metadata_enabled = True
enable_distributed_floating_ip = True
```

Once OVN is configured, further network resource creation and management will be handled by OVN, which maps each OpenStack resource abstraction to an entry object in the OVN southbound and northbound databases.

Both OVS and OVN provide different capabilities to connect instances to networks. In the following section, we will explore how routing is performed in Neutron and demonstrate the different routing traffic options in OpenStack.

Configuring cloud routing

Instances within the same virtual tenant network can reach each other, but by default, each tenant network cannot reach other tenants or external networks. Deploying virtual routers is the way to enable L3 network communication so that tenant virtual networks can connect by associating a subnet with a router.

Routing tenant traffic

Under the hood, a port associated with a tenant virtual network will be associated with the IP address of the subnet gateway. Instances across different virtual networks reach each other by communicating via the virtual router, using the gateway IP address and their private IP addresses encapsulated in the packets. This is called a **NAT** (**network address translation**) mechanism. In OpenStack networking, the Neutron L3 agent manages virtual routers. IP packets are forwarded by a virtual router to different self-service and external networks through the following router interfaces:

- `qr`: Contains the tenant network gateway IP address and is dedicated to routing traffic through all self-service traffic networks
- `qg`: Contains the external network gateway IP address and is dedicated to routing traffic out onto the external provider network

Upon the initiation of a virtual router instance, a network namespace will be created in the Neutron node that defines connections to self-service or external provider networks through routing tables, packet forwarding, and `iptables` rules. As shown in the following diagram, a router namespace refers to a virtual router that is attached to multiple bridge ports in an OVS configuration (**qr** and **qg**). Traffic flow between instances hosted in the same or different compute nodes is routed through the virtual router.

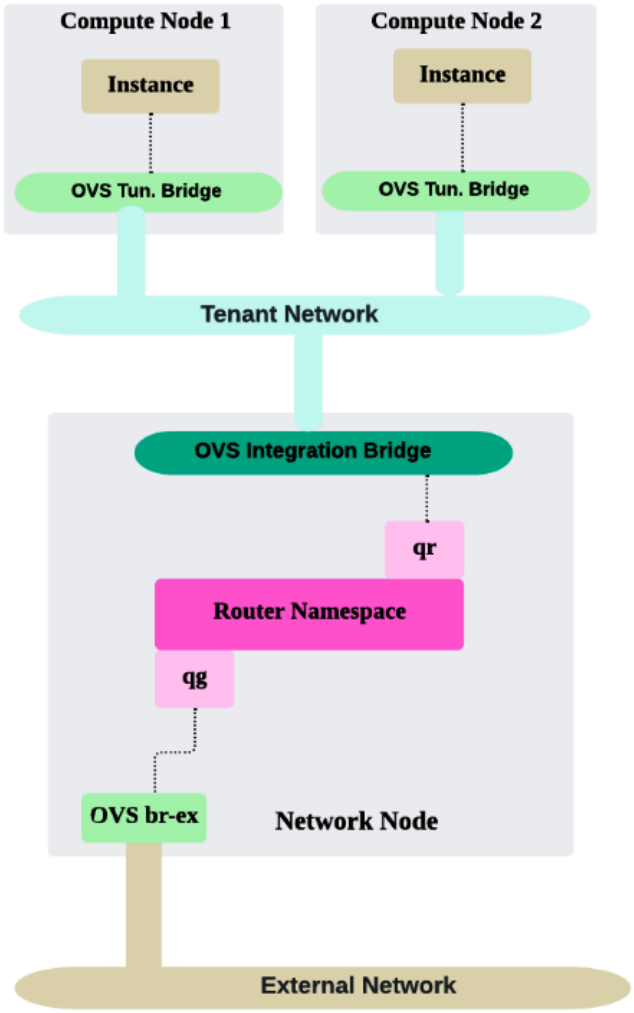


Figure 6.8 – Neutron router namespace connectivity based on OVS implementation

In the previous OVS implementation, the L3 agent should be up and running to start creating and managing virtual routers.

Important note

The router service plugin should be enabled by default once Neutron agent L3 is deployed, using `kolla-ansible`. The service plugin can be verified in the `/etc/neutron/neutron.conf` file in the Neutron cloud controller node by checking the `service_plugin = router` line.

Optionally, virtual routers can be managed through Horizon. The `kolla-ansible` run for the Neutron deployment should enable the router module in the dashboard. That can be verified in the cloud controller node's `/etc/openstack-dashboard/local_settings.py` file with the following settings:

```
...
OPENSTACK_NEUTRON_NETWORK = {
    'enable_router': True,
...
```

In the next exercise, we will create a tenant virtual network with the ML2 plugin configured in OVS. A virtual router will be attached to the tenant network using the OpenStack CLI, as follows:

1. Create a tenant virtual network using the OpenStack network CLI:

```
$ openstack network create network_pp
```

2. Create a subnet with an IP range of `10.10.0.10/24`, an auto-assigned DHCP, and a default DNS nameserver of `8.8.8.8`:

```
$ openstack subnet create --subnet-range 10.10.0.0/24 --network
network_pp --dns-server 8.8.8.8 priv_subnet
```

3. Create a router and attach it to the created tenant network:

```
$ openstack router create router_pp
$ openstack router add subnet router_pp priv_subnet
```

4. The router attachment to the tenant network will assign a private IP address to the router's internal interface. By default, if no IP address is specified in the attachment command line, the internal interface will be assigned the default gateway of the subnet. The following command line verifies the assigned IP address of the router's internal interface:

```
$ openstack port list --router router_pp
```

It gives the following output:

```
+--------------------------------------+------+-------------------+-------------------------------------------------------------------------------+--------+
| ID                                   | Name | MAC Address       | Fixed IP Addresses                                                            | Status |
+--------------------------------------+------+-------------------+-------------------------------------------------------------------------------+--------+
| 423fd213-443a-dc33-ba3d-bd3a236264d2 |      | fa:16:3e:90:44:6a | ip_address='10.10.0.1', subnet_id='e34a22a1-77a1-4e31-ffac-30304201e3e3'      | ACTIVE |
+--------------------------------------+------+-------------------+-------------------------------------------------------------------------------+--------+
```

Figure 6.9 – Virtual router port listing

5. The router interface can be checked within the router namespace `qr-` prefixed interface by running the following command:

```
$ ip netns
```

The output looks like the following:

```
qdhcp-76da2e37-d5de-11cd-cdd1-fe3adf8cdaea
qrouter-3a211622-11da-9687-bda1-acae3d74ad12
```

Figure 6.10 – The network namespaces list

6. Copy the `qrouter` ID from the previous output, and then run the following command line to show the created interface of the router and the assigned IP address from the internal network:

```
$ ip netns exec qrouter-3a211622-11da-9687-bda1-acae3d74ad12 ip
addr show
```

We will get the following output:

```
   ...
2: qr-473f51c5-47: < BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether fa:16:3e:33:6d:d7 brd ff:ff:ff:ff:ff:ff
    inet 10.10.0.1/24 brd 10.10.0.255 scope global qr-473f51c5-47
    ...
```

Figure 6.11 – A virtual router namespace internal interface listing

To enable instances to reach the external network, a second interface in the virtual router should be created that will be attached to the provider's external network. A common networking setup, such as an external network device or a device integrated with a firewall appliance, should be placed in front of the OpenStack endpoint (such as the load balancer). Enabling internet access can be configured using the OpenStack CLI, as follows:

1. Create an external network provider. As configured in the ML2 plugin for OVS, we can create the network as a VLAN type, with a defined segmentation ID and `physnet1` for the physical network attribute:

```
$ openstack network create --external --provider-network-type
vlan --provider-segment 40 --provider-physical-network physnet1
external_pp
```

2. Create the subnet part of the external network with a network range of `10.20.0.0/24` and the default gateway as `10.20.0.1`, disable DHCP, and set an allocation pool of `10.20.0.10-10.20.0.100`:

```
$ openstack subnet create --subnet-range 10.20.0.0/24 --no-dhcp
--network external_pp --allocation-pool start=10.20.0.10,
end=10.20.0.100 pub_subnet
```

3. Attach the router to an external provider network by running the following command line:

```
$ openstack router set --external-gateway external_pp router_pp
```

4. The attachment operation will assign an external IP from the external IP pool to the router, which can be checked by running the following command line:

```
$ openstack port list --router router_pp
```

Here is the output:

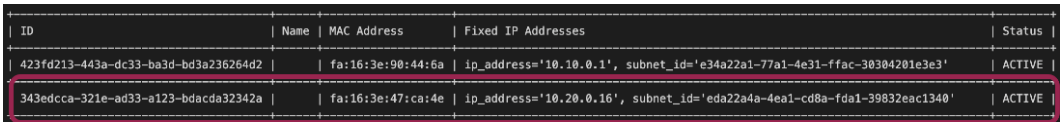


Figure 6.12 – A created external port of the virtual router

5. The last attachment creates a second interface, prefixed with `qg-` in the router namespace, which can be verified by the following command line:

```
$ ip netns exec qrouter-3a211622-11da-9687-bda1-acae3d74ad12 ip
addr show
```

Here is the output:

```
3: qg-e3a36727-f1: < BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether ed:17:ae:32:5d:e8 brd ff:ff:ff:ff:ff:ff
    inet 10.20.0.16/24 brd 10.20.0.255 scope global qg-e3a36727-f1
```

Figure 6.13 – A virtual router namespace external interface listing

The next part of our connectivity demonstration involves creating security groups and rules to allow ingress and egress traffic at the network port level. To reach the internet and access the instances, we will create a new security group and add **ICMP** and **SSH** access:

1. Using the OpenStack CLI, create a new security group to be applied to your instances:

```
$ openstack security group create SG_pp
```

2. Create the rules associated with the created security group for SSH and ICMP, respectively:

```
$ openstack security group rule create SG_pp --protocol tcp
--dst-port 22
$ openstack security group rule create SG_pp --protocol icmp
```

3. Using the OpenStack CLI, create a new test instance with a `tiny` flavor and a `cirros` image that is connected to the private tenant network:

```
$ openstack server create --flavor tiny --image cirros-0.5.2
--nic net-id=network_pp --security-group SG_pp instance_pp
```

Important note

Make sure to adjust your available Glance image name and list of flavors, based on your existing resources. The `openstack server create` command line will fail if any of the assigned arguments do not exist. Cirros is a minimal Linux distribution, useful for quick testing and proof of concepts. The default session username is `cirros` and the password is `gocubsgo`.

4. To test the connectivity from the instance to reach the internet, make sure that the instance state is **ACTIVE**, as follows:

```
$ openstack server list
```

Here is the output:

```
+--------------------------------------+-------------+--------+----------------------+--------------+----------+
| ID                                   | Name        | Status | Networks             | Image        | Flavor   |
+--------------------------------------+-------------+--------+----------------------+--------------+----------+
| fd2eras2-bd31-3211-8723-ca44e3a187a5 | instance_pp | ACTIVE | network_pp=10.10.0.12 | cirros-0.5.2 | tiny     |
+--------------------------------------+-------------+--------+----------------------+--------------+----------+
```

Figure 6.14 – The instance listing

5. Access to the created instance can be made in different ways, using the `virsh console` command line from the compute node or simply via SSH from the router namespace. Make sure to use the default `cirros` image credentials – that is, the username `cirros` and the password `gocubsgo` – and run a simple ping to reach `8.8.8.8`:

```
$ ip netns exec qrouter-3a211622-11da-9687-bda1-acae3d74ad12 ssh
cirros@10.10.0.12
```

Here is the output:

```
login as 'cirros' user. default password: 'gocubsgo'. use 'sudo' for root.
instance_pp login: cirros
Password:
$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=112 time=14.658 ms
```

Figure 6.15 – Testing external connectivity

6. The instance uses the internal virtual router IP address as the default gateway to route traffic reaching the external network. The internet is reachable through SNAT, performed by the router. A quick run of the `ip route` command in the instance shows the default gateway associated with the network route table:

```
$ ip route
default via 10.10.0.1 dev eth0
169.254.169.254 via 10.10.0.1 dev eth0
10.10.0.0/24 dev eth0 src 10.10.0.12
```

Figure 6.16 – Listing the default gateway

The next part of our walk-through is a demonstration of how resources hosted in external networks can reach instances in an OpenStack environment. By default, spawned instances will be assigned an IP address that is not visible outside of the tenant network. Neutron provides float IP addresses that implement **DNAT** (**destination NAT**). The router simply forwards incoming packets reaching its external interface to the destination instance by checking its configured DNAT rule. The traffic response from an instance to external resources uses the source IP addresses translated to the floating IP, demonstrated as follows:

1. Extract the port ID of the created instance, and create a floating IP address to associate with it:

```
$ openstack port list --server instance_pp
```

Here is the output:



Figure 6.17 – Listing the instance port

2. Copy the port ID, and run the following command line by pasting the external port ID after the `--port` option:

```
$ openstack floating ip create --port 524ead12-33da-dc11-e3a1-
dc34e6da1c81 external_pp
```

Here is the output:

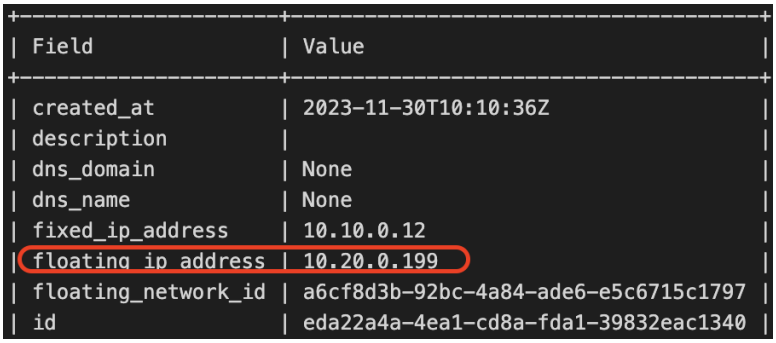


Figure 6.18 – Assigning the floating IP address to the external port

3. Under the hood, the router namespace configures a secondary address associated with the external interface, prefixed with `'qg'`:

```
$ ip netns exec qrouter-3a211622-11da-9687-bda1-acae3d74ad12 ip
addr show
```

Here is the output:

```
3: qg-e3a36727-f1: < BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen
    link/ether ed:17:ae:32:5d:e8 brd ff:ff:ff:ff:ff:ff
    inet 10.20.0.16/24 brd 10.20.0.255 scope global qg-e3a36727-f1
       valid_lft forever preferred_lft forever
    inet 10.20.0.199/32 brd 10.20.0.199 scope global qg-e3a36727-f1
```

Figure 6.19 – Associating the IP address with the virtual router external interface

Traffic routed through the external network provider can reach the instance via the assigned floating IP.

So far, we have explored a standard routing implementation in OpenStack. The next section will uncover another way of performing routing, via dynamic routing, in Neutron.

Neutron dynamic routing

Dynamic routing in OpenStack networking is based on **BGP** (**Border Gateway Protocol**), enabling tenant networks to advertise their network prefixes with physical or virtual routers and network devices that support BGP. The new addition of BGP in Neutron eliminates the usage of floating IPs for tenant networks that do not rely on network administrators to advertise their network upstream. The term *dynamic routing* in Neutron was introduced with the **Mitaka** release. The adoption of the BGP routing mechanism varies from one cloud environment to another, depending on the networking setup, mostly due to the requirement for direct connectivity between the network node and the physical network gateway device to peer with (such as a LAN or WAN peer). To avoid IP overlapping when advertising IP prefixes, dynamic routing relies on **address scopes** and **subnet pools**, Neutron mechanisms that control the allocation of subnet addresses and prevent the usage of addresses that overlap.

At the heart of the BGP implementation, Neutron introduced the **BGP speaker**, which enables peering between tenant networks and external router devices. The BGP speaker advertises the tenant network to the tenant router, initially as a first hop. The BGP speaker in Neutron is not a router instance, nor does it manipulate BGP routes. It mainly orchestrates the BGP peer's information between the tenant routers and external ones. The speaker requires a network or cloud operator to configure the peering endpoints. As shown in the following diagram, for a successful BGP dynamic routing in OpenStack, the Neutron virtual router must be attached to both the tenant subnet and the external provider device interface.

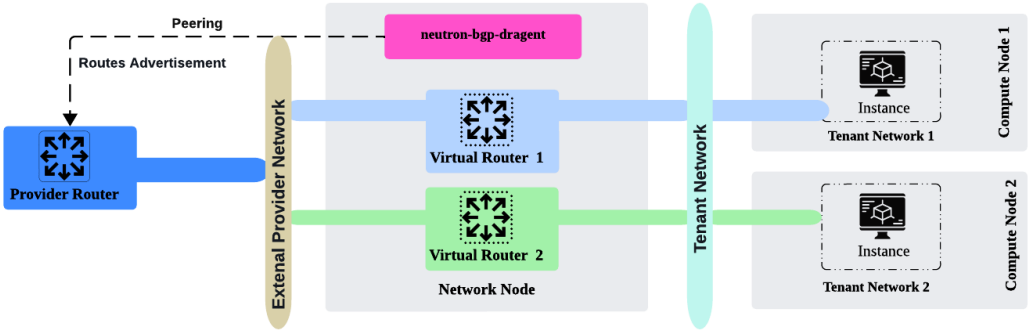


Figure 6.20 – Neutron BGP peering and router connectivity for dynamic routing

Both the BGP speaker and the external provider device must be peered (connected to the Neutron virtual router). Finally, both the tenant and the external networks must be in the same address scope. Adding BGP dynamic routing using `kolla-ansible` is straightforward. We will configure the dynamic routing based on the OVS implementation.

> **Important note**
>
> Since the Antelope release, Neutron has supported dynamic routing with the OVN mechanism driver.

As Neutron provides a BGP agent with default configuration, we will just need to enable the agent installation in the network node by adding the following line to the `/ansible/inventory/multi_packtpub_prod` inventory file:

```
...
[neutron-bgp-dragent:children]
neutron
```

Enable the agent installation in the `globals.yml` file, as follows:

```
enable_neutron_bgp_dragent: "yes"
```

Launch the pipeline to roll out the BGP agent installation in the network node. In the network node, the installed BGP agent can be checked by running the following command line:

```
$ openstack network agent list --agent-type bgp
```

Here is the output:

```
+--------------------------------------+---------------------+---------+-------------------+-------+-------+---------------------+
| ID                                   | Agent Type          | Host    | Availability Zone | Alive | State | Binary              |
+--------------------------------------+---------------------+---------+-------------------+-------+-------+---------------------+
| 43911e32-4411-4de3-34a1-ca32e17d1fe2 | BGP dynamic routing agent | net01.os | None              | :-)   | UP    | neutron-bgp-dragent |
+--------------------------------------+---------------------+---------+-------------------+-------+-------+---------------------+
```

Figure 6.21 – Listing the BGP network agent

The Neutron BGP CLI to manage BGP speakers can be found at `https://docs.openstack.org/python-openstackclient/latest/cli/plugin-commands/neutron.html`.

Virtual routers are part of the building blocks of OpenStack networking, providing a variety of connectivity options to tenants. It is important to note that running with a standalone router presents a single point of failure. *Chapter 7*, *Running a Highly Available Cloud – Meeting the SLA*, will discuss Neutron implementation for highly available routers. Dynamic routing via BGP in Neutron is considered an amazing routing addition in OpenStack. Neutron has been enriched and modified across OpenStack releases. One of these major modifications is the development of new networking services in OpenStack, which will be discussed in the next section.

Joining more networking services

Besides the overwhelming routing and switching capabilities in Neutron that come with the latest OpenStack releases, Neutron enables cloud operators to offer additional networking services. Services such as load balancing, firewalls, and private networks support tenant networks to build well-architected application stacks for a variety of use cases. In the next section, we will cover the most used and stable additional Neutron services and deploy them, using `kolla-ansible`.

Load balancer as a service

Since the **Liberty** release, the Neutron **Load Balancer as a Service** (**LBaaS**) plugin has moved from version 1 to version 2, codenamed **Octavia**. Cloud users who seek to balance the load of their workloads using a managed load balancer will appreciate the Octavia offering. Octavia is designed to scale horizontally, as it creates and manages virtual machines acting as load-balancer instances, referred to as **amphorae**.

Octavia implements the same load-balancing terms as the popular **HAProxy**, which can be summarized as follows:

- **Virtual IP (VIP)**: A layer 4 object that exposes a service to be accessed, externally or internally, that is associated with a Neutron port. A load balancer is assigned a VIP, and requests that reach the VIP are distributed among backend servers or pool members.
- **Pool**: A group of instances serving the same content or service, such as web servers.
- **Pool members**: An instance of a pool represented by an IP address and listening port that exposes the service.
- **Listeners**: A port associated with the VIP that listens for incoming requests.
- **Health monitor**: Orchestrates the management of pool members based on health checks carried out on each member. Failed health checks will discard the members from the pool, and traffic will be served by the healthy ones.
- **Load-balancing algorithms**: **Round robin**, **least connections**, and **source IPs** are supported algorithms in Octavia that can be assigned when creating a pool.
- **Session persistence**: A mechanism that forces client requests to be served by the same backend server. Since the depreciation of LBaaS v1, the load-balancing service in Neutron can be configured through drivers. The most common ones are HAProxy and Octavia. Other third-party vendor drivers can also be integrated, such as F5 and Citrix, and Neutron will manage the orchestration of the API calls in the OpenStack environment. A glossary of different components in Octavia is as follows:
    - **Amphora**: A virtual machine acting as a load balancer, running on compute nodes and preconfigured with load-balancing parameters, such as pools, backend members, and health monitors.

- **Controller worker**: Updates the load-balancer instance (Amphora instance) with configuration settings.
- **API controller**: Interacts with the controller worker for load-balancer (Amphora instances configurations and operations for deployment, deletion, and monitoring.
- **Health manager**: Watches the state of each load balancer (Amphora instance) and triggers failover events during the unexpected failure of a load balancer.
- **Housekeeping manager**: Removes stale database records and handles the spare pool.

The deployment of Octavia using `kolla-ansible` would require more steps than just installing an agent. To install the load-balancing service in OpenStack using the Octavia driver, start by assigning the core Octavia components to run as part of the cloud controller:

```
...
[octavia:children]
control
[octavia-api:children]
octavia
[octavia-driver-agent:children]
octavia
[octavia-health-manager:children]
octavia
[octavia-housekeeping:children]
octavia
[octavia-worker:children]
octavia
```

Enable the Octavia service in the `globals.yml` file, as follows:

```
...
enable_octavia: "yes"
```

Make sure that in the `globals.yml` file, the `enable_neutron_provider_networks` setting is set to `true`. This is because we need the Octavia nodes to communicate through the management network. Assign the network interface of the `eth0` management network of the cloud controller node:

```
octavia_network_interface: eth0
```

`kolla-ansible` automates most of the Octavia deployment and registration of the associated resources in the OpenStack environment. Octavia interacts with several OpenStack services, including Nova and Neutron. Associated resources can be customized in the `globals.yml` file, such as the amphorae Nova flavor, security groups, and the Octavia network and subnets, which can be set by updating `octavia_amp_flavor`, `octavia_amp_security_groups`, and the **Octavia management network**, respectively. By default, `kolla-ansible` is configured to register all

dependent Octavia resources automatically. If you want to customize the Octavia registration service, make sure you set all required values for each Octavia setting in the `globals.yml` file. That can be done after changing the following setting from `no` to `yes`:

```
octavia_auto_configure: "yes"
```

Run the deployment pipeline to roll out the Octavia service. The deployment run should create the necessary Octavia resources, including certificates, the Octavia network, the flavor, the security group, and the SSH key. Make sure that `kolla-ansible post-deploy` runs successfully by generating an `octavia-openrc.sh` file, loaded as follows:

```
$ kolla-ansible post-deploy
$ . /etc/kolla/octavia-openrc.sh
```

Octavia uses images to launch amphora instances. Run the following command lines to download a recent amphora image for the Bobcat release, and then add it to Glance:

```
$ wget https://swift.services.a.regiocloud.tech/swift/v1/AUTH_
b182637428444b9aa302bb8d5a5a418c/openstack-octavia-amphora-image/
octavia-amphora-haproxy-2023.2.qcow2
$ openstack image create amphora-haproxy --container-format bare
--disk-format qcow2 --private --tag amphora --file octavia-amphora-
haproxy-2023.2.qcow2 --property hw_architecture='x86_64' --property
hw_rng_model=virtio
```

The following steps assume that two web server instances named `instance1` and `instance2` have been deployed and are connected to the tenant network used in a previous demonstration, `10.10.0.0/24`:

1. Note two of the created instances:

   ```
   $ openstack server list
   ```

   Here is the output:

```
+--------------------------------------+-----------+--------+--------------------------+---------------+----------+
| ID                                   | Name      | Status | Networks                 | Image         | Flavor   |
+--------------------------------------+-----------+--------+--------------------------+---------------+----------+
| ead33e23-98da-3341-cda1-763ee212da39 | instance1 | ACTIVE | network_pp=10.10.0.114   | cirros-0.5.2  | tiny     |
| 01dca123-ddca-6531-443a-6218daca2384 | instance2 | ACTIVE | network_pp=10.10.0.125   | cirros-0.5.2  | tiny     |
+--------------------------------------+-----------+--------+--------------------------+---------------+----------+
```

Figure 6.22 – Listing the created instances

2. To quickly mimic a load-balancing exercise, run the `SimpleHTTPServer` Python module in each of the instances. Create a simple `index.html` file to trace the load-balancer member pool usage. On `instance1`, run the following:

   ```
   $ echo "Reaching instance 1" > ~/index.html
   $ python -m SimpleHTTPServer
   ```

3. Run the same command lines by specifying the second instance in the `index.html` file:

```
$ echo "Reaching instance 2" > ~/index.html
$ python -m SimpleHTTPServer
```

4. Make sure to allow ingress port `80` on the instances by creating and adding the following security group and rule, if not already attached:

```
$ openstack security group create lb-web-sg
$ openstack security group rule create --ingress --protocol http
--dst-port 80 --ethertype IPv4 lb-web-sg
$ openstack server add security group instance1 lb-web-sg
$ openstack server add security group instance2 lb-web-sg
```

5. Create a load balancer attached to the private subnet, `priv_subnet`. Note that the load balancer will be assigned a VIP that will be used to expose the backend service and allow clients to access it:

```
$ openstack loadbalancer create --name lb --vip-subnet-id
priv-subnet
```

Here is the output:

```
+---------------------+--------------------------------------+
| Field               | Value                                |
+---------------------+--------------------------------------+
| admin_state_up      | True                                 |
| availability_zone   | None                                 |
| created_at          | 2023-12-01T10:18:32                  |
| description         |                                      |
| flavor_id           | None                                 |
| id                  | c4352a23-651a-ddea-65e1-f5341627daac |
| listeners           |                                      |
| name                | lb                                   |
| operating_status    | OFFLINE                              |
| pools               |                                      |
| project_id          | cda334e5dcfaa23165241fd23d6212ca     |
| provider            | amphora                              |
| provisioning_status | PENDING_CREATE                       |
| updated_at          | None                                 |
| vip_address         | 10.10.0.14                           |
| vip_network_id      | 1348de46-4411-e3e3-cde3-52425e425625 |
| vip_port_id         | cdaa312d-65da-547a-87dd-de3da45a12a8 |
| vip_qos_policy_id   | None                                 |
| vip_subnet_id       | e34a22a1-77a1-4e31-ffac-30304201e3e3 |
+---------------------+--------------------------------------+
```

Figure 6.23 – Creating the load balancer

6. Once the load balancer updates its `PROVISIONING_STATUS` state from `PENDING_CREATE` to `ACTIVE` and `OPERATING_STATUS` to `ONLINE`, create a TCP listener port named `listener80` on port `80`:

```
$ openstack loadbalancer listener create --name listener80
--protocol TCP --protocol-port 80 lb
```

7. Create a load-balancing pool named `poolweb`, and specify the load-balancing algorithm with `ROUND_ROBIN`:

```
$ openstack loadbalancer pool create --name poolweb
--lb-algorithm ROUND_ROBIN --listener listener1 --protocol TCP
```

8. Add a health monitor to the created `poolweb` with the backend servers and probe the HTTP service port:

```
$ openstack loadbalancer healthmonitor create --name http-check
--delay 15 --max-retries 4 --timeout 30 --type HTTP poolweb
```

9. Join the backend instances (`instance1` and `instance2`) to `poolweb`:

```
$ openstack loadbalancer member create --subnet-id private-
subnet --address 10.10.0.114 --protocol-port 80 poolweb
$ openstack loadbalancer member create --subnet-id private-
subnet --address 10.10.0.125 --protocol-port 80 poolweb
```

10. Optionally, check the status of the added backend members and created pool:

```
$ openstack loadbalancer member list poolweb
```

Here is the output:

id	name	project_id	provisioning_status	address	protocol_port	operating_status	weight
ead33e23-98da-3341-cda1-763ee212da39		cda334e5dcfaa23165241fd23d6212ca	ACTIVE	10.10.0.114	80	ONLINE	1
01dca123-ddca-6531-443a-6218daca2384		cda334e5dcfaa23165241fd23d6212ca	ACTIVE	10.10.0.125	80	ONLINE	1

Figure 6.24 – Listing the instance members of the load-balancer pool

11. Test the load balancer by connecting to its VIP. Note that requests will be served consecutively by both instances, as the algorithm used for load balancing is round robin:

```
$ for ((i=1;i<=10;i++)); do curl http://10.10.0.14; sleep 1;
done
```

Here is the output:

```
Reaching instance 1
Reaching instance 2
Reaching instance 1
Reaching instance 2
Reaching instance 1
Reaching instance 1
...
```

Figure 6.25 – Testing the load-balancer pool backend

Important note

Accessing the pool web backend from the internet can be done in different ways. One way is to attach the load balancer to a public network directly without the need to run behind a virtual router, and the load balancer VIP will be assigned a public routable IP. The other way is by creating a floating IP, and then associating it with the VIP of the load balancer that exists within a subnet behind the virtual router. A floating IP cannot be routable through the internet without a router that applies IP translation, using the assigned floating IP, to a public one to reach external networks.

A good practice when creating load balancers in Neutron is to think about security first. The latest LBaaS version in OpenStack supports TLS certificate deployment. Even better, OpenStack has a dedicated project that handles keying and certificates, codenamed *Barbican*. To learn more about Barbican, refer to the official OpenStack docs: `https://docs.openstack.org/barbican/latest/`. A load balancer can work in tandem with the Barbican service to handle its TLS certificates most securely.

Summary

OpenStack networking provides an overwhelming list of features that enable cloud operators and architects to implement advanced network topologies and offer cloud users and tenants more managed network services. As seen in this chapter, the ML2 plugin extends Neutron's capabilities by unlocking some pertinent features that were lacking in older OpenStack releases. OVS was explored as a mechanism driver that brings more traffic control based on flow rules. For extended networking that requires common VLAN and VXLAN topologies, the chapter demonstrated the support of OVS and OVN drivers in Neutron within the latest OpenStack releases, in addition to their deployments using `kolla-ansible`. It is worth noting that the direction of future networking based on mechanism drivers is almost certainly moving toward adopting the OVN mechanism driver, due to its performance in larger production-grade and hybrid cloud environments. Another big deal, routing in the OpenStack networking world, was discussed. This chapter also revisited a routing implementation using OVS and covered BGP dynamic routing. OVN is becoming a popular way to implement BGP , and new releases might show more stable integration in that direction. However, the chapter did not cover all

other extra Neutron services and agents, such as **VPN as a service**, **firewall as a service**, and **Designate** for DNS management. LBaaS in Neutron has seen more updates with the latest OpenStack releases. With the new LBaaS v2, cloud operators can configure their load-balancing service by using open source drivers, such as HAProxy or Octavia. Finally, the chapter demonstrated the deployment of a load-balancer setup using Octavia, which is becoming a more widely adopted LBaaS solution that is considered suitable for large-scale production environments.

In the next chapter, we will enhance our deployed OpenStack environment by tackling resiliency and high availability for the OpenStack control and data planes.

Part 2: Operating the OpenStack Cloud Environment

This part will spot the operational mission of cloud operators to bring the cloud to the next stage. Essential operational excellence pillars will be covered, including high availability, monitoring, and logging, to ensure business continuity and keep an eye on common best practices on cloud health. This part will conclude with an exclusive addition on setting up a continuous infrastructure evaluation and resource optimization layouts.

This part has the following chapters:

- *Chapter 7*, *Running a Highly Available Cloud – Meeting the SLA*
- *Chapter 8*, *Monitoring and Logging – Remediating Proactively*
- *Chapter 9*, *Benchmarking the Infrastructure – Evaluating Resource Capacity and Optimization*

7

Running a Highly Available Cloud – Meeting the SLA

"The past resembles the future more than one drop of water resembles another."

– Ibn Khaldun

One major aspect of successful cloud operating experiences is to prevent downtime and the failures of cloud resources and workloads. In *Chapter 1*, *Revisiting OpenStack – Design Consideration*, we drafted an initial preparatory design to enable OpenStack services for redundancy. *Chapter 3*, *OpenStack Control Plane – Shared Services*, and *Chapter 4*, *OpenStack Compute – Compute Capacity and Flavors*, looked at some of the logical design patterns for OpenStack control plane deployments and various ways to segregate compute, such as cells and availability zones. OpenStack is designed to scale massively and providing hardware for dedicated OpenStack services can help isolate failures, but this requires mechanisms to keep services running during incidents.

Ensuring **high availability** (**HA**) in the OpenStack world does not differ too much from any other complex IT system. One obligatory practice is to find and eliminate, through the logical OpenStack setup, any possible **single points of failure** (**SPOFs**), which differ from one design to another. Our goal in this chapter is to achieve HA in each layer of the private cloud infrastructure.

In this chapter, we will cover the following topics:

- Reviewing HA and failover strategies to ensure business continuity
- Iterating through OpenStack control plane HA design patterns
- Deploying an OpenStack environment with additional cloud controllers for fault tolerance and redundancy using `kolla-ansible`
- Exploring different ways to achieve networking HA in Neutron with the latest OpenStack updates, including routing and distributed virtual router mechanisms
- Uncovering native OpenStack solutions to ensure instances failover using the Masakari OpenStack project

Exploring HA strategies

A robust OpenStack cloud platform includes fault tolerance at every level of its architecture. This can be very successful if it is planned in advance. Starting with a small cluster is easy and achievable, but growing it is a challenge. The hallmark of the basic OpenStack component is that it can run on commodity hardware. OpenStack is designed to scale massively and provide HA by leveraging more advanced HA techniques at each level of the infrastructure. This can include **automatic failover** and **geo-redundancy**. *Chapter 4*, *OpenStack Compute – Compute Capacity and Flavors*, introduced the concepts of cells, regions, and availability zones, which provide more robust and advanced fault tolerance and availability capabilities for massive OpenStack deployment at scale.

Measuring HA

Service availability should be measured and defined by standard metrics. This can be summarized using the following formula:

Availability = MTTF/ (MTTF + MTTR)

In the preceding equation, we can see the following:

- **Mean time to failure (MTTF)**: An estimate of the average time that a system is functional before its failure
- **Mean time to repair (MTTR)**: An estimate of the average time to repair a part or component of a system

Measuring HA in complex environments such as OpenStack would require a good understanding of the deployed cloud environment capabilities that can be tracked through performance metrics and KPIs, such as response time, system uptime, and downtime, for the **Repair Time Objective** (**RTO**) and **Repair Point Objective** (**RPO**). What's more critical to end users is to expose a **Service-Level Agreement** (**SLA**) from the gathered metrics and KPIs and improve against them. A SLA identifies areas of improvement based on routinely gathered metrics and will boost your business continuity

strategy. Availability management is an integral pillar of IT best practices that cannot be skipped, especially when operating a cloud environment running dozens of services, as with OpenStack. Creating those SLAs for each service in more depth would fill an entire book. For the sake of simplicity, make sure you engage in availability management practices and update the SLA per service by assigning an availability level and availability and downtime percentages, as shown in the following table:

Service	**Availability Level**	**Availability**	**Downtime/Day**
Compute	One 9	90	~ 2.4 hours
Network	Two 9s	99	~ 14 minutes
Compute	Three 9s	99.9	~ 86 seconds
Block storage	Four 9s	99.99	~ 8.6 seconds
Object storage	Five 9s	99.999	~ 0.86 seconds
Image	Six 9s	99.9999	~ 0.0086 seconds

Table 7.1 – Example SLA with x-9s for OpenStack environment services

When designing the architecture for an HA OpenStack setup, failures should be planned for at every single layer of the cloud architecture. This can be achieved thanks to advanced HA models and techniques. The latest OpenStack releases are even richer, with built-in features that embrace availability not only for core components but also for user workloads. For example, if a host fails, the application running on it will not be accessible anymore. Nova supports the feature to recover a guest instance by relocating it to a new healthy host. For an extended OpenStack setup, the cloud environment could scale even more domain failures by recalling availability zones under the Nova service. The essence of the HA design patterns can be summarized in the following points:

- Eliminate any SPOF for the control and data planes
- Adopt a geo-replicated design whenever possible
- Automate monitoring and anomaly detection
- Plan and automate fast disaster recovery
- Decouple and isolate OpenStack components as much as possible

In the world of OpenStack, different levels of HA can be identified:

- **L1**: This includes physical hosts, network and storage devices, and hypervisors.
- **L2**: This includes OpenStack services, including compute, network, and storage controllers, as well as databases and message queuing systems.
- **L3**: This includes the virtual machines running on hosts that are managed by OpenStack services.
- **L4**: This includes applications running in the virtual machines themselves.

The main focus of supporting HA in OpenStack is on L1, L2, and L3.

Designing for HA

One key aspect when designing systems is to take into account every possibility where an element of the system could fail. Each element has limits of some kind and would not be able to recover within at least a short period, which would affect other parts of the system and lead to the whole system becoming unresponsive. When looking at common design patterns aimed at maximizing scalability and availability, it is important to identify two main classes of services:

- **Stateful service**: A service that depends on data from a previous request and interacts synchronously to preserve consistency. As such services rely on the state, a service failure can affect the whole system and require more backup and recovery mechanisms to maintain state information.
- **Stateless service**: A service that does not require data or needs to save information (state) from a previous request or event. As such services do not store states across requests, a sudden failure won't affect the rest of the system, which can operate in a different handler instance.

As discussed in *Chapter 3*, *OpenStack Control Plane – Shared Services*, the OpenStack control plane is mainly constructed of stateless services, including an API, scheduler, agents, and conductor components for all compute, network, image, monitoring, and storage services. The database and queuing message count as stateful services. This way, introducing HA into our initial setup would require verifying the right pattern for each service and component of the OpenStack deployment to ensure the continuity of the service during unexpected failures. There are a multitude of ways to achieve OpenStack control plane HA. This can be summarized in terms of the following design patterns, both of which were introduced in *Chapter 3*, *OpenStack Control Plane – Shared Services*:

- **Active/passive**: In the OpenStack control plane, failed services will be restarted on a second cloud controller node. With stateful services such as the database, the master node handles all read and write operations and a second node acts as a listener until a failover occurs, upon which the data entry point will be switched to the second node. This pattern is suitable for some stateful services but not for other stateless OpenStack services. Additionally, it does not fully comply with horizontal scaling in OpenStack.
- **Active/active**: Here, the request load hitting the OpenStack control plane is distributed among active nodes that process in parallel. This mode in an OpenStack deployment brings the highest level of fault tolerance for the control and data plane services. In the event of one cloud controller failure, the rest of the load will be distributed to the second operational node, maintaining uninterrupted services. It is also suited to horizontal scaling as it adds a new node to accommodate any increased load without compromising the function of the cluster. As OpenStack core components are based on API calls and RPC to handle messages via the queue message system, performance is paramount. In active/active mode, service recovery has a shorter MTTR than active/passive mode, where a potentially long delay to failover can occur, causing some OpenStack services to time out.

Most of the OpenStack reference architectures adopt an active/active ground setup for fault tolerance and failover, mainly due to the nature of the OpenStack services that can scale horizontally easily and do not require additional mechanisms. On the other hand, some services can be configured in active/passive mode, depending on the nature of the service. In the next section, we will look at the different services that can be used to enable HA.

Preparing for HA

Each OpenStack control plane layer requires separate analysis to identify the best approaches to ensure its availability. Additional technical considerations will be taken into account when designing for fault tolerance in a production setup. It is important to note that a major factor in the different choices comes from the given cloud service provider's experience with tooling or hardware solutions. In this section, we will go through some of the major adopted tools and design patterns to achieve a highly available and scalable OpenStack control plane.

Designing with load balancing

Load balancing solution services can be found everywhere and can be used to efficiently distribute and serve incoming requests across a server pool. HAProxy has been widely adopted on dozens of large OpenStack production deployments.

The HAProxy setup involves the following two types of servers:

- **Frontend server**: This server listens for requests coming from a specific IP and port, and determines where the connection or request should be forwarded
- **Backend server**: A pool of servers in the cluster receiving the forwarded requests

It is also important to note the function layers that are involved in HAProxy load balancing:

- **Layer 4**: Load balancing is performed in the transport layer in the OSI model. All the user traffic will be forwarded based on a specific IP address and port to the backend servers. For example, a load balancer might forward the internal OpenStack system's request to the Horizon web backend group of backend servers. To do this, whichever backend Horizon is selected should respond to the request under scope. This is true in the case of all the servers in the web backend serving identical content.
- **Layer 7**: The application layer will be used for load balancing. This is a good way to load balance network traffic. Simply put, this mode allows you to forward requests to different backend servers based on the content of the request itself.

HAProxy supports several load balancing algorithms to dispatch a request to a server in the backend pool:

- **Round robin**: Each server is exploited in turn. As a simple HAProxy setup, round robin is a dynamic algorithm that defines the server's weight and adjusts it on the fly when the called instance hangs or starts slowly.
- **Leastconn**: The selection of the server is based on the node that has the lowest number of connections.
- **Source**: This algorithm ensures that the request will be forwarded to the same server based on a hash of the source IP, so long as the server is still up.
- **Uniform resource identifier** (**URI**): This ensures that the request will be forwarded to the same server based on its URI. It is ideal to increase the cache hit rate in the case of proxy cache implementations.

HAProxy keeps an eye on the nodes' backend availability by running health checks on particular IP addresses and ports. It disables any backend node that fails the health checks and discards it from the backend pool until it is healthy enough to start serving requests again.

Armed with a load balancer, an OpenStack service that is deployed in two or more nodes will be exposed by a **Virtual IP** (**VIP**). In active/active mode, the VIP is managed by the load balancer, which makes sure that a node has sufficient availability before it forwards the request.

> **Important note**
>
> If you consider separating the HAProxy deployment into its own physical setup, make sure that the VIPs can be reached over the public network.

As the use of VIPs combined with HAProxy adds an extra layer to protect OpenStack services from failures, the load balancing layer should not present a single point of failure. Depending on which software or hardware-based load balancing solution you adopt, make sure you increase its redundancy level. That should be reviewed as a critical networking setup as it defines the first interface of the OpenStack environment. This can be achieved by using a VIP software management tool such as **Keepalived** or **Pacemaker** to ensure a highly available load balancing layer. Keepalived is free software and employs the **Virtual Router Redundancy Protocol** (**VRRP**) to eliminate SPOFs by making IPs highly available. As shown in the following diagram, VRRP implements virtual routing to perform failover tasks between two or more servers in a static, default-routed environment:

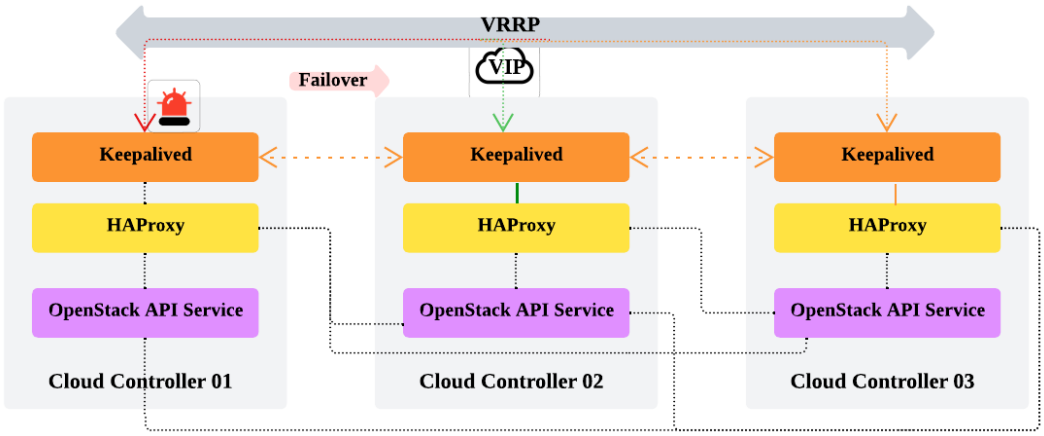


Figure 7.1 – Load balancing and failover using HAProxy and Keepalived

> **Important note**
>
> To ensure an easily determined quorum by Keepalived, we will empower our control plane with HA via three cloud controller nodes managed by Keepalived.

HA for the database

Databases have always been a critical subject when it comes to dealing with the growth of data being stored, along with performance and resiliency. This is because a database is not a simple service and does not scale as fast as a simple API. Any request that reaches an OpenStack API service results in the database's size growing incrementally. In our deployment process, using a CI/CD pipeline will generate a few additional entries across several tables in each run. If not designed to scale and monitored closely, the database could be subject to failure and quickly become a bottleneck. Several open source and database vendors provide possible topologies to scale horizontally or vertically or both. The other factor is the type of database engine supported by OpenStack and the required experience of the cloud operation team to build and architect a highly available and scalable database solution in OpenStack.

Since we've started our initial production draft with a single database based on MySQL, we can highlight the most commonly clustering topologies:

- **Master/slave replication**: A VIP will be switched to a slave node when the master node fails. A delay in the health check on the master node at failover time and thus a delay in assigning the VIP to the slave node could potentially result in data inconsistencies, as shown here:

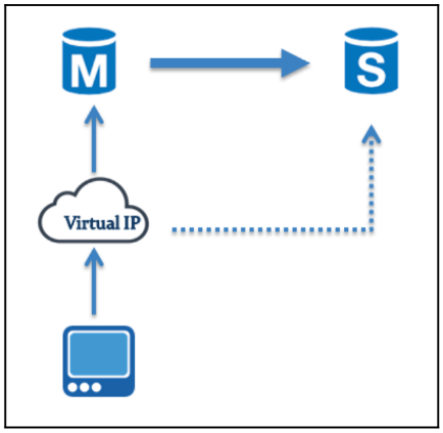


Figure 7.2 – Database master-slave replication

- **Multi-master replication manager** (**MMM**): By setting up two servers, both of them become masters by keeping only one acceptable write query at a given time. This is still not a very reliable solution for OpenStack database HA because in the event of failure of the master, it might lose a certain number of transactions, as illustrated here:

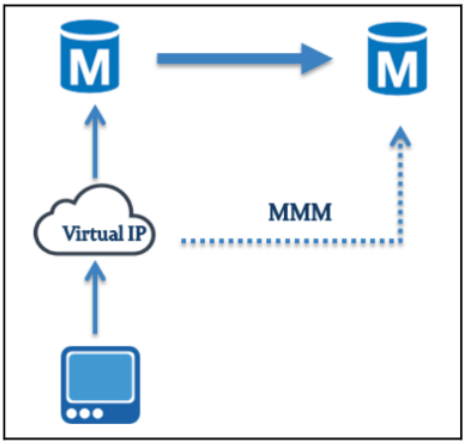


Figure 7.3 – Database MMM replication

- **MySQL shared storage**: In this topology, both servers depend on redundant shared storage. As shown in the following figure, a separation is required between the servers processing the data and the storage devices. Note that an active node may exist at any point in time. If it fails, the other node will take over the VIP after checking the inactivity of the failed node, and turn it off. Such a solution is excellent in terms of uptime but may require a powerful storage/hardware system, which can be extremely expensive. The service will be resumed on a different node by mounting the shared storage within the taken VIP:

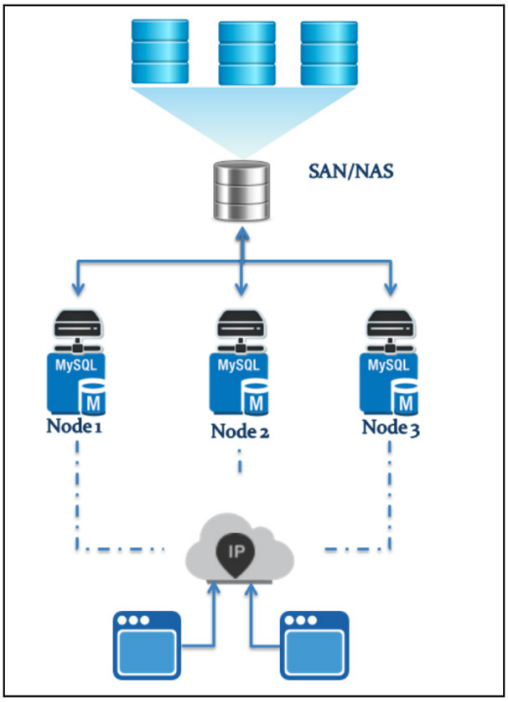


Figure 7.4 – Database MySQL with shared storage

- **Block-level replication**: One of the most adopted HA implementations is the **Distributed Replicated Block Device** (**DRBD**) replication. Simply put, it replicates data in the block device, which is the physical hard drive that's shared between OpenStack MySQL nodes. DRBD can be a costless solution, but performance-wise, it is not sufficient when you're relying on hundreds of nodes. It can also affect the scalability of the replicated cluster:

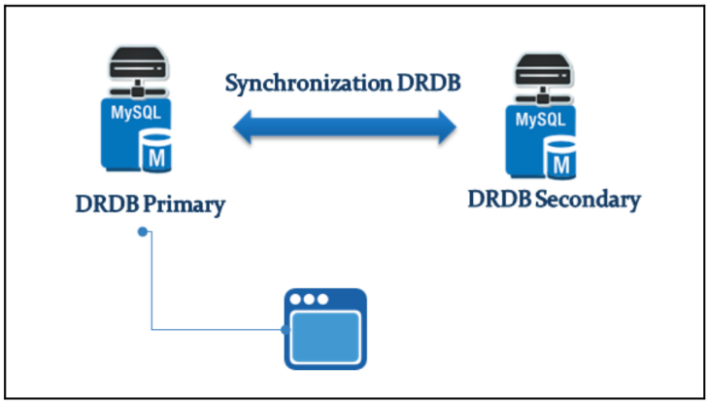


Figure 7.5 – Database block-level replication

- **MySQL multi-master replication with Galera**: Based on **multi-master replication**, the **Galera** solution has a few performance challenges within an MMM architecture for MySQL/InnoDB database clusters. A requirement for the Galera setup to run properly is the presence of at least three nodes. As shown in the following diagram, synchronous replication is managed by Galera, where data is replicated across the whole cluster:

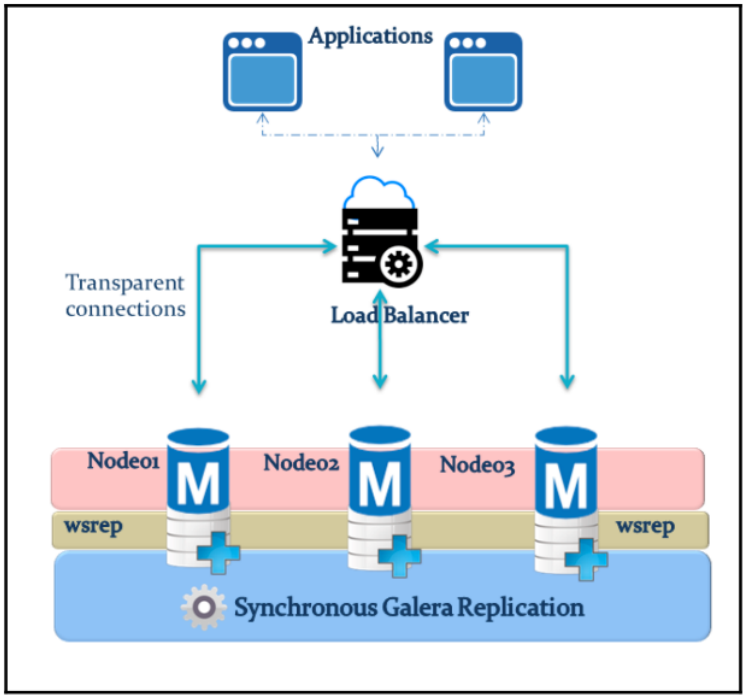


Figure 7.6 – Database multi-master with Galera replication

Regarding these topologies, any MySQL replication setup can be simple to set up and make HA-capable, but data can be lost during the failover process. MySQL multi-master replication with Galera is tightly designed to resolve such a conflict in the multi-master database environment. An issue you may face in a typical multi-master setup is that all the nodes try to update the same database with different data, especially when a synchronization problem occurs during the master failure. This is why Galera uses **certification-based replication** (**CBR**). The main mechanism of CBR is to assume that the database can roll back uncommitted changes and is transactional, in addition to applying replicated events in the same order across all the instances. Replication is truly parallel; each one has an ID check. The added value that Galera can bring to our MySQL (MariaDB in OpenStack) HA is the ease of scalability, such as joining a node to Galera in an automated fashion in a production environment. The end design brings an active/active multi-master topology with minimum latency and transaction losses. A best practice to ensure data consistency when implementing Galera for MariaDB in OpenStack is to keep writes committed to only one node of the three.

HA for the message bus

RabbitMQ is mainly responsible for communication between different OpenStack services. The issue is fairly simple: no queue, no OpenStack service intercommunication. RabbitMQ should be considered another critical service that needs to be available and survive failures.

> **Important note**
>
> A variety of queuing messages systems such as Qpid or ZeroMQ are mature enough to support their own cluster setup without the need for you to run other resource managers or clustering software solutions alongside them.

RabbitMQ is a robust messaging system that can achieve scalability in an active/active way through one of the following patterns:

- **Clustering**: Any data or state needed for the RabbitMQ broker to be operational is replicated across all nodes.
- **Mirrored queues**: As the message queue cannot survive in the nodes in which it resides, RabbitMQ can act in active/active HA message queues. Simply put, queues will be mirrored on other nodes within the same RabbitMQ cluster. Thus, any node failure will lead to automatically switching to one of the queue mirrors.
- **Quorum queues**: This is a modern version of queues using a variant of the **Raft** protocol (enabling members of a distributed system to agree on a set of values and share data in the event of failure) with a distributed consensus algorithm and implementing replicated FIFO. Each quorum queue has a leader and multiple followers with replicated queues hosted in different hosts.

RabbitMQ has deprecated the implementation of mirrored queues in favor of quorum queues due to the problems of its predecessor that it solves. This includes synchronization failing and performance issues. With the huge message bus traffic in OpenStack, quorum queues can boost not just availability but also the consistency of the messages, as shown in the following diagram:

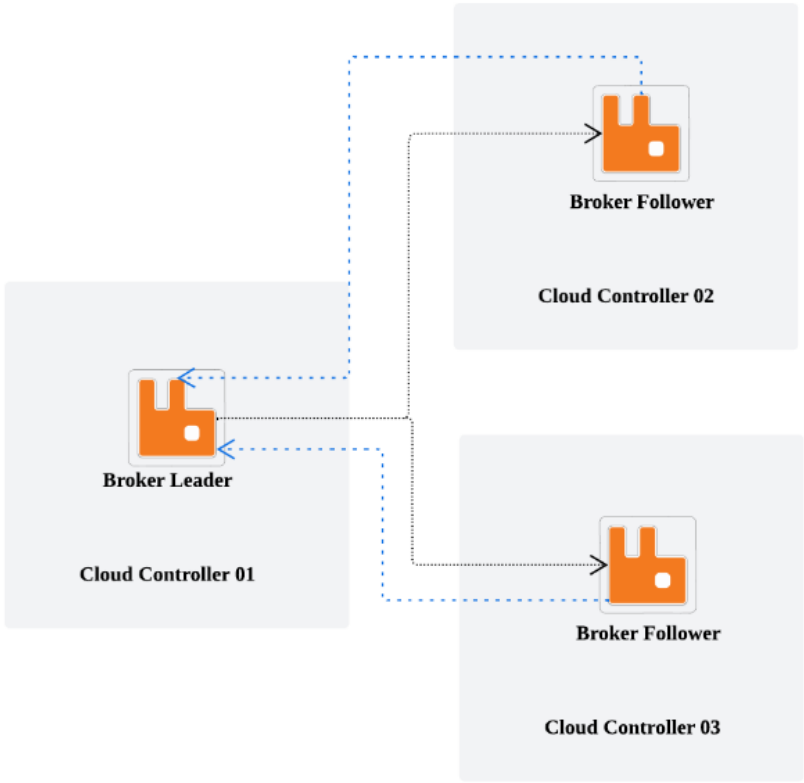


Figure 7.7 – RabbitMQ broker quorum pattern

In the next section, we will extend our deployment by introducing the aforementioned elements to enable HA and redundancy in the OpenStack environment.

Deploying for HA

In this section, we will extend our initial production deployment by extending our OpenStack control plane with a highly available configuration composed of the following set of nodes:

- **Virtual IP**: `10.0.0.47`
- **HAProxy 01** (`hap1.os.packtpub`): `10.0.0.20`
- **HAProxy 02** (`hap2.os.packtpub`): `10.0.0.21`
- **Cloud Controller 01** (`cc01.os.packtpub`): `10.0.0.100`
- **Cloud Controller 02** (`cc02.os.packtpub`): `10.0.0.101`
- **Cloud Controller 03** (`cc03.os.packtpub`): `10.0.0.102`

The HA version of our OpenStack environment will require that we apply the following configurations in the `globals.yml` file:

- Enable HAProxy, which will use Keepalived by default for the HA settings:

```
enable_haproxy: "yes"
```

- Assign a VIP that is not used for the management network where the HAProxy hosts are connected and Keepalived is running. Optionally, the external and internal VIPs can be separated. The following setting will use the same internal VIP address:

```
kolla_external_vip_address: "10.0.0.47"
```

- The RabbitMQ quorum is the default implementation for message-queue HA in OpenStack. Some older versions in `kolla-ansible` use mirrored queues referenced with the `om_enable_rabbitmq_high_availability` setting. Make sure this is disabled and uses quorum queues instead by checking the `ansible/group_vars/all.yml` file or adding the following variable set if it does not exist in the `globals.yml` file:

```
om_enable_rabbitmq_quorum_queues: true
```

Important note

Populating the quorum queues in a running environment requires manually restarting all OpenStack services using the `kolla-ansible stop --tags <service-tags>` and `kolla-ansible deploy --tags <service-tags>` command lines, where `<service-tags>` is the name of a given OpenStack service. It is recommended to automate the service restart using the pipeline for consistent configuration and durable queues.

The next configuration update is to adjust the `multi_packtpub_prod` file. The following layout suggests the deployment of three cloud controller nodes:

```
...
[control]
cc01.os.packtpub
cc02.os.packtpub
cc03.os.packtpub
```

A new host group of two load balancers will be added, running HAProxy and Keepalived:

```
...
[haproxy]
hap1.os.packtpub
hap2.os.packtpub
[loadbalancer:children]
haproxy
```

As stated in *Chapter 3*, *OpenStack Control Plane – Shared Services*, one of the best practices regarding production environments is to start hosting workloads in the cloud environment only when it is configured with HA (at a minimum) within its core services. As no production workload has been run yet, it is recommended to clean up the running environment by firing off the following command line:

```
$ kolla-ansible -i ./multi_packtpub_prod destroy --yes-ireally-really-
mean-it
```

This will clean up the OpenStack service containers and associated volumes. The new pipeline run will deploy all containers from the same image.

The database deployment for Galera InnoDB will run the `wsrep` service across the three controller nodes. One common issue when deploying additional nodes in a running environment is the failure of one or both nodes to read binary logs and update the replication status. RabbitMQ quorum also requires additional manual tweaks to clean the existing exchanges and move to durable queues while using quorum across different cloud controller nodes.

Commit the changes before running the pipeline. Rolling out the new multi-node infrastructure will take longer than the very first run as the new cloud controller nodes and load balancers will be deployed in addition to multi-master database quorum queues deployment across different nodes.

Once the pipeline finishes the multi-node deployment, observe the Docker images that have been loaded for HAProxy and Keepalived by running the following command line in any of the controller nodes:

```
$ sudo docker images
```

We will get this output:

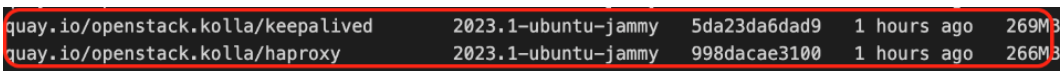


Figure 7.8 – Listing the HAProxy and Keepalived Kolla images

In addition to the different containers for OpenStack services, observe the container running HAProxy and Keepalived:

```
$ sudo docker ps
```

Here is the output:



Figure 7.9 – Listing the HAProxy and Keepalived Kolla containers

Optionally, verify that all compute nodes can be listed as part of the OpenStack environment and check the service status, as follows:

```
$ openstack availability zone list --compute --long
```

The output is as follows:

```
+-----------+-------------+---------------+-------------+----------------+--------------------
| Zone Name | Zone Status | Zone Resource | Host Name   | Service Name   | Service Status
+-----------+-------------+---------------+-------------+----------------+--------------------
| internal  | available   |               | cc01.os     | nova-scheduler | enabled :-) 2023-11
| internal  | available   |               | cc02.os     | nova-conductor | enabled :-) 2023-11
| internal  | available   |               | cc03.os     | nova-scheduler | enabled :-) 2023-11
| internal  | available   |               | cc03.os     | nova-conductor | enabled :-) 2023-11
| internal  | available   |               | cc02.os     | nova-scheduler | enabled :-) 2023-11
| internal  | available   |               | cc01.os     | nova-conductor | enabled :-) 2023-11
| nova      | available   |               | cc01.os     | nova-compute   | enabled :-) 2023-11
| nova      | available   |               | cc02.os     | nova-compute   | enabled :-) 2023-11
+-----------+-------------+---------------+-------------+----------------+--------------------
```

Figure 7.10 – Listing the enabled Nova services in all OpenStack environments

Each HAProxy instance that's deployed in each cloud controller node is assigned a priority ID that's used by Keepalived to refer to the elected master node. The generated file in each controller node can be found in the `/etc/kolla/keepalived/keepalived.conf` file. The following is a snippet of the Keepalived configuration that was generated on one of the cloud controller nodes:

```
vrrp_instance kolla_internal_vip_51 {
    state BACKUP
    nopreempt
    interface br0
    virtual_router_id 51
    priority 40
    advert_int 1
    virtual_ipaddress {
        10.0.0.47 dev br0
    }
}
...
```

The `kolla_internal_vip_51` configuration block defines a random unique ID for the VIP that can be set in the `globals.yml` file by changing the `keepalived_virtual_router_id` variable. The default value that's shown is `51`, which refers to the `virtual_router_id` value in the Keepalived configuration. Once a cluster managed by Keepalived is launched, a priority number will be assigned for each node, where the higher priority is the most preferred node to hold the VIP and hence is elected as the master. In this example, a priority of `40` has been assigned to the current

cloud controller node. A quick check on the Kolla Keepalived container log is used to validate each cloud controller state. In the following example, the current cloud controller is assigned a master state:

```
$ docker logs -f keepalived
```

And we get the following output:

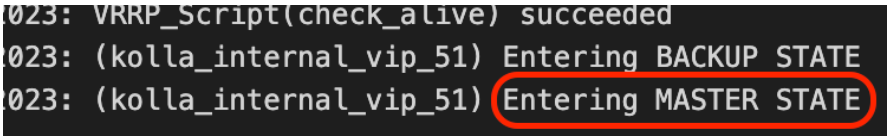


Figure 7.11 – Validating the Keepalived master assignment

Upon being elected as a master, Keepalived assigns a VIP of `10.0.0.47` to the cloud controller node. In this example, cloud controller 02 (`10.0.0.101`) has been assigned the VIP. This can be checked by firing off the following command line in the host:

```
$ ip a
```

It gives the following output:

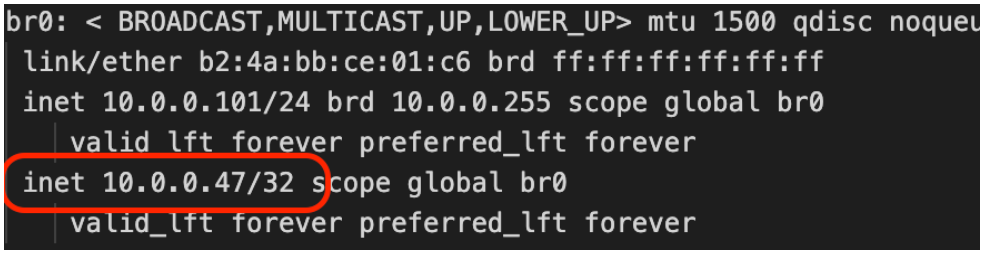


Figure 7.12 – Checking the Keepalived VIP

Enabling HAProxy and Keepalived in our OpenStack deployment ensures HA for most OpenStack services. On the other hand, OpenStack networking may require additional hardening to enable the fault tolerance capability, as will be depicted in the following subsection.

HA for networking

The OpenStack network service involves different composites, including Neutron server L2, L3, metadata, and DHCP agents. L2 agents are installed on every compute node and there is no need to maintain their HA setup. The DHCP and metadata agents run across multiple nodes and support a highly available setup by default.

On the other hand, L3 agents require more tweaking to attain HA as they are the ones responsible for the following aspects:

- Managing virtual routers per tenant
- Providing external connectivity to instances
- Managing floating IPs to instances for external network access

Before the **Icehouse** release, there was no built-in solution to resolve the L3 agent HA issue. Some workarounds involve utilizing an external cluster solution using Pacemaker and **Corosync**. New HA modes adopted for Neutron in OpenStack have been introduced since the **Juno** release, including the following options:

- **Virtual Router Redundancy Protocol** (**VRRP**)
- **Distributed Virtual Routing** (**DVR**)

The next few sections will look at how a redundant Neutron router setup can be achieved using VRRP and Keepalived.

Routing redundancy with VRRP

We briefly introduced the concept of VRRP earlier in this chapter. In the networking context, VRRP and Keepalived are configured in Neutron to fail over within a brief period between router namespaces. As shown in the following diagram, routers can be seen in the form of groups, where each group presents an active router that is currently forwarding traffic to instances:

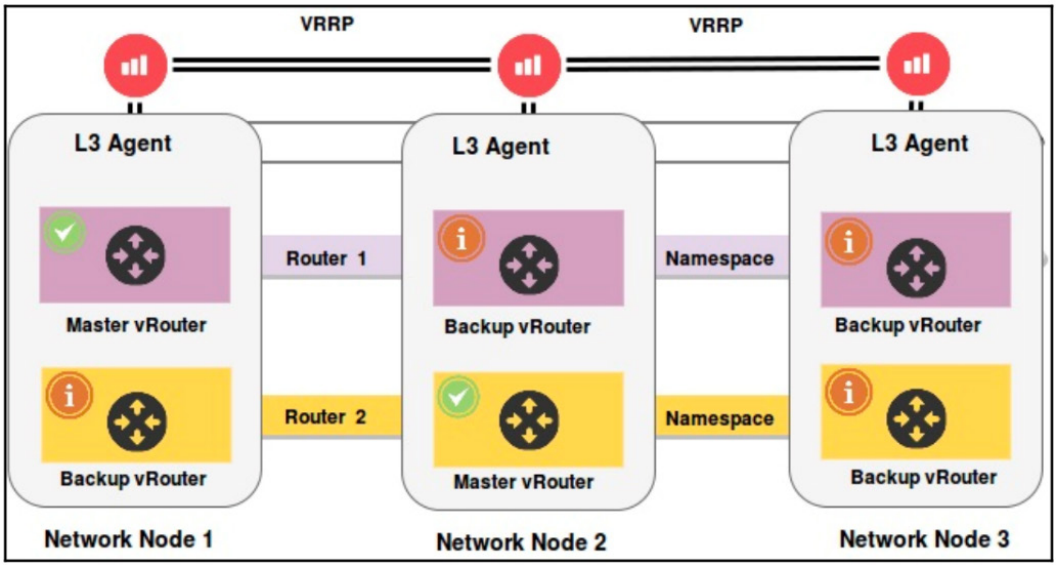


Figure 7.13 – Routing redundancy with VRRP

Additionally, the instance traffic is spread among all network nodes based on the scheduling for the master and the rest of the backup routers. Based on the same concept of the Keepalived mechanism, the master router configures its VIP internally and keeps informing the router group about its state and priority.

Each VRRP group elects a master router based on the assigned priorities, where the router with the highest ID will be selected as the master and the rest remain as backups. A new election poll will only take place when the master router stops sending its VRRP advertisements to the assigned group and will be marked as a failed master.

Every newly created HA router will add a new router namespace where its L3 agent starts Keepalived. Under the hood, routers configured in HA mode will be able to communicate via a specific HA network that is not visible to users. The HA network interface is denoted by `ha`.

> **Important note**
>
> The active router needs to periodically inform the standby ones about its state, determined by the **advertisement interval timer**. If a backup router does not receive such information, it will start a new master router election process based on the last advertised VRRP. This election is based on priority, where the router with the highest value will be elected as the master. Priorities range from `0` to `255`, with `255` being the highest priority.

By default, Neutron automatically creates an HA pool range network of `169.254.192.0/18`, which is used by tenant routers with HA mode enabled. The next configuration demonstrates a router resiliency setup in Neutron using VRRP and Keepalived. Based on our initial design, a Neutron node will be added that runs an L3 agent and is configured with the Open vSwitch mechanism driver, which supports HA routers.

Add a second network node to the inventory file that runs an L3 agent and, optionally, the DHCP and metadata agents, as follows:

```
...
[network]
net02.os.packtpub
...
```

Run the pipeline and make sure the L3 agent is up and running by running the following command line on the controller node:

```
$ openstack network agent list --agent-type l3
```

The output will be as shown:



id	agent_type	host	alive	admin_state_up	binary
3ab3a7e5-bbf9-4ac8-8b9f-2044e021cf23	L3 agent	net01.os	:-)	True	neutron-l3-agent
23ca2234-e3ea-87da-ca33-fda23ca7da11	L3 agent	net02.os	:-)	True	neutron-l3-agent

Figure 7.14 – Listing the installed Neutron L3 agents

Important note

Make sure the new network node is connected to the same network segment as the first network node, as illustrated in *Chapter 1*, *Revisiting OpenStack – Design Considerations*. The second node will use the same Neutron mechanism driver for Open vSwitch, as configured in the `globals.yml` file.

Set the following configuration to `yes` in the `globals.yml` file to enable the HA state for the Neutron L3 agent:

```
enable_neutron_agent_ha: "yes"
```

Run the pipeline to apply the new changes in the Neutron configuration file. The Neutron service should be restarted, with the `l3_ha` setting being set to `True` in the `/etc/neutron/neutron.conf` file. The default number of maximum L3 agents that will be scheduled on a virtual router is set to `3` to construct the VRRP virtual router. This can be modified in the `roles/neutron/defaults/main.yml` file by setting the value of `max_l3_agents_per_router`.

With these new settings, any newly created router is considered an HA router and not a legacy one. Create a new router by setting the `–ha` flag:

```
$ openstack router create --ha haRouter
```

Walking through the different network nodes running an L3 agent, the newly created router can be observed via its created namespace in the first network node:

```
@net01.os:~#ip netns | grep router
qrouter-ba14211-22ae-3422-cda1-aeb623dacd11
And on the second network node:
@net02.os:~#ip netns | grep router
qrouter-ba14211-22ae-3422-cda1-aeb623dacd11
```

Once an HA router is created, the master router will be assigned a virtual IP of `169.254.0.1` at any given time:

```
$ ip netns exec qrouter-ba14211-22ae-3422-cda1-aeb623dacd11 ip addr
show
```

The output will be as follows:

```
...
5: ha-da420251-b4@if21: < BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default qlen 1000
    link/ether fa:16:3e:79:c1:11 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 169.254.192.1/18 brd 169.254.255.255 scope global ha-da420251-b4
       valid_lft forever preferred_lft forever
    inet 169.254.0.1/24 scope global ha-da420251-b4
       valid_lft forever preferred_lft forever
```

Figure 7.15 – Listing the router namespaces and HA scope for the master router

Let's check the output in the second node running the backup router:

```
$ ip netns exec qrouter-ba14211-22ae-3422-cda1-aeb623dacd11 ip addr
show
```

We get the following:

```
...
2: ha-6a6af123-3a@if7: < BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default qlen 1000
    link/ether fa:16:3e:18:69:74 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 169.254.192.2/18 brd 169.254.255.255 scope global ha-6a6af123-3a
       valid_lft forever preferred_lft forever
```

Figure 7.16 – Listing the router namespaces and HA scope for the backup router

Neutron automatically reserves a new dedicated HA network that is only visible to administrators and does not belong to any OpenStack project upon the creation of HA routers:

```
$ openstack network list
```

The output is shown as follows:

```
+--------------------------------------+-----------------------------------------------------+--------------------------------------+
| ID                                   | Name                                                | Subnets                              |
+--------------------------------------+-----------------------------------------------------+--------------------------------------+
| ba223a13-43ea-33ac-872a-bac123665da2 | HA network tenant 67df4262ddf241a7626524da4c986911  | 7623da12-1ea2-ccda-987a-cdf3a4da8986 |
| 523daca6-22ac-70da-9861-ddca34198dca | network_pp                                          | 432eda21-887d-aad7-9851-44da8dae1ba9 |
| dac21234-2341-98da-3312-ca7d53a9dfac | external_pp                                         | 9834daca-1123-33da-123a-4dac317812ca |
+--------------------------------------+-----------------------------------------------------+--------------------------------------+
```

Figure 7.17 – Listing the reserved HA network

Keepalived is configured to run in each namespace by using a persistent configuration file located at `/var/lib/neutron/ha_confs/ROUTER_NETNS/keepalived.conf`, where `ROUTER_NETNS` is the router namespace of the HA router. Failover events are also logged in `neutron-keepalived-state-change.log` under the same directory. The following extract from a log shows a switch router during a failover event on the first network node:

```
...
DEBUG neutron.agent.l3.keepalived_state_change [-] Wrote router
qrouter-ba14211-22ae-3422-cda1-aeb623dacd11 state master write_state_change
...
```

Figure 7.18 – Verifying router failover in log entries

Neutron routing fault tolerance is a critical requirement to handle OpenStack networking availability. The OpenStack Neutron design was not limited only to router redundancy using VRRP. Later, Neutron introduced the DVR implementation. We'll explore this in the next subsection.

Routing redundancy with DVR

Like HA routers, DVR operates across multiple compute nodes. Using DVR, network load is distributed across operating routers, reducing the traffic load on the network node. L3 agents run in compute nodes and traffic flows for *east-west* (instance-to-instance) and *north-south* (from an instance to external networks with floating IPs or vice versa) are routed across them instead of a single centralized network node.

Configuring DVR in Neutron requires the usage of the Open vSwitch mechanism driver and the L3 agent to be installed on the compute nodes. Update the inventory file by adding the L3 agent to the desired compute nodes, as follows:

```
...
[compute]
cn01.os.packtpub
cn02.os.packtpub
...
[neutron-l3-agent:children]
compute
```

Run the pipeline and verify that the L3 agent is up and running in the compute nodes by running the following command line on the controller node:

```
$ openstack network agent list --agent-type l3
```

We get the following output:

```
+--------------------------------------+----------------+-------------+-------+----------------+--------------------+
| id                                   | agent_type     | host        | alive | admin_state_up | binary             |
+--------------------------------------+----------------+-------------+-------+----------------+--------------------+
| 4232eaa2-d311-34a1-4433-265da7c81963 | L3 agent       | cn01.os     | :-)   | True           | neutron-l3-agent   |
| 872da232-eac4-98dd-caef-fd4ed111872d | L3 agent       | cn02.os     | :-)   | True           | neutron-l3-agent   |
+--------------------------------------+----------------+-------------+-------+----------------+--------------------+
```

Figure 7.19 – Listing the installed Neutron L3 agents

Enable DVR routing in the `globals.yml` file by setting the following configuration line:

```
enable_neutron_dvr: "yes"
```

Run the pipeline and observe the configuration updates in the `/etc/neutron/neutron.conf` Neutron configuration file, as follows:

```
...
router_distributed = True
```

Each compute node with an installed L3 agent should host the `/etc/neutron/plugins/ml2/openvswitch_agent.ini` Open vSwitch agent configuration file with the following settings:

```
...
[agent]
l2_population = True
enable_distributed_routing = True
```

Run the pipeline to populate the DVR configuration in both the agent and Neutron server configuration settings.

As an admin, create a new router by specifying the `--distributed` argument, as follows:

```
$ openstack router create --distributed router_dvr
```

In each compute node, validate the existence of the same `qrouter` namespace ID:

```
root@cc01.os:~#ip netns | grep router
qrouter-a312143-ea11-3417-adc1-1eb513daeeda
root@cc02.os:~#ip netns | grep router
qrouter-a312143-ea11-3417-adc1-1eb513daeeda
```

The router connecting both networks to different compute nodes is the same router instance with the same namespace created in each compute node. Typically, the `qr` interfaces corresponding to the same namespace will have the same interface names and IP addresses in the compute nodes. For `east-west` connectivity, the traffic flow between `instance01` hosted in `cn01.os` reaching `instance02` spawned in `cn02.os` can be visualized as follows:

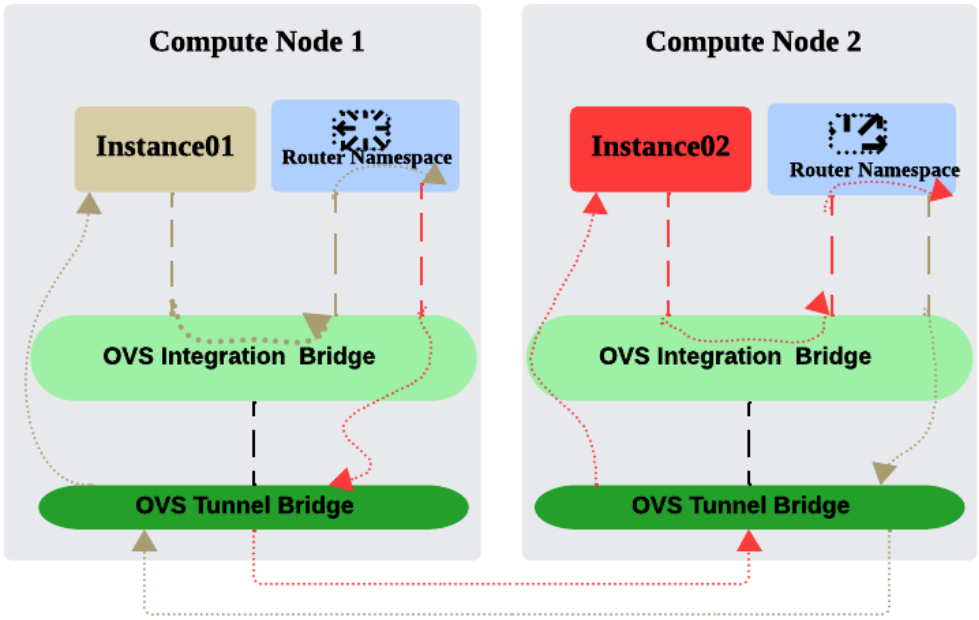


Figure 7.20 – East-west traffic in DVR mode

As shown in the preceding diagram, traffic flowing from `instance01` and `instance02` (east-west) takes the following path:

1. Traffic from `instance01` is forwarded from its local gateway through the router namespace.
2. The router in `cn01.os` replaces the source MAC address with its interface MAC address.
3. The router forwards the traffic to the integration bridge in `cn01.os`.
4. Packets are then forwarded to the provider bridge in `cn01.os`.
5. The source MAC address of the router interface in the packet is replaced by the MAC address of the compute node, `cn01.os`.
6. Traffic is forwarded to the destination compute node, `cn02.os`, through the physical network.
7. Traffic reaches the `cn02.os` host via the provider bridge.
8. Traffic is forwarded to the integration bridge in the `cn02.os` host.
9. At the integration bridge level, the source MAC address is replaced by the router MAC interface in `cn02.os`, as dictated in the router namespace.

10. Traffic is forwarded to the destination, `instance02`.
11. Return traffic from `instance02` follows the routing path from `cn02.os` through its respective bridges and routers.

The DVR implementation, much like using VRRP, is simple as it comes with Neutron's built-in features.

This section outlined the building blocks of enabling HA in the underlying OpenStack infrastructure and control plane services. In the following section, we will deep dive into the different options to implement fault tolerance for workloads running in OpenStack.

Managing instance failover

Providing HA to cloud resources is a critical topic that OpenStackers have had to handle since the early releases of OpenStack. Workload users seek different ways to increase the availability of their virtual machines through manual tooling and scripting, which can be overlooked if not efficiently managed. With the introduction of the **Masakari** project in OpenStack, cloud operators can offer workload users an automated service that ensures HA for KVM-based instances, reducing the need for manual scripting and seamlessly integrating within the OpenStack ecosystem.

Masakari also uses Corosync and Pacemaker. As a native HA load balancing stack solution for the Linux platform, Pacemaker is a cluster resource manager that depends on Corosync to control and organize HA across the hosts in OpenStack. Corosync ensures that cluster communication is based on the messaging layer and manages the VIP assignment to one of the nodes. Once a cluster of workload instances is created, Masakari provides failure detection of the hosts running the instances, which is where Corosync comes into play by making sure a virtual IP is assigned to a functional host. Masakari is composed of an API that deals with REST requests and an engine component that executes recovery requests to the Nova service. As shown in the following figure, Masakari primarily provides instance HA via three forms of monitors:

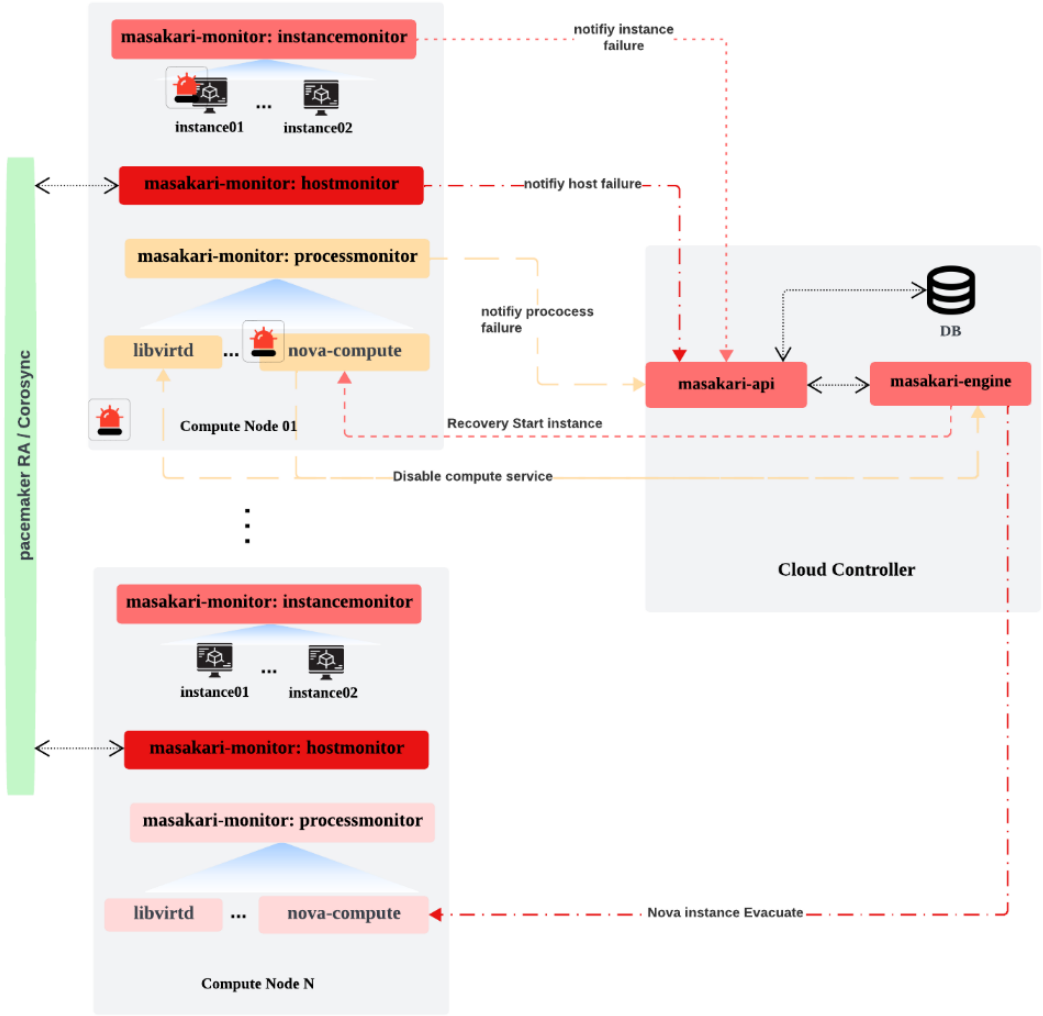


Figure 7.21 – Instance HA monitors using Masakari

The main Masakari components can be summarized as follows:

- **Instance restart**: An instance failure detected by an agent running in the compute node will be restarted. Instance restart is managed by the `masakari-instancemonitor` Masakari monitor process.
- **Instance evacuation**: Instances will be evacuated to another healthy compute node upon detection of hypervisor failure. The Masakari glossary defines a **failover segment** as a group of compute nodes that hosts evacuated instances if one of the compute nodes in the same segment goes down. Instance evacuation is managed by the `masakari-hostmonitor` Masakari process monitor.

- **Process monitor**: This is managed by the `masakari-processmonitor` service, which runs on the compute node to collect the status of the different processes running on the hypervisor machine, including libvirtd and the Nova compute service. The monitor makes sure that, during the failure of one process, no more instances are scheduled to run on the affected compute node and will instead be handled by other nodes.

The next section describes how to deploy the Masakari service in OpenStack using `kolla-ansible`.

Deploying Masakari

Masakari is composed of its API, monitors, and engine, all of which are installed across the cloud controller and compute nodes. Starting with the `globals.yml` file, enable the Masakari service by editing the following variable:

```
enable_masakari: "yes"
```

To control the instances' HA using the instance evacuation option, enable the following setting in the `globals.yml` file:

```
enable_hacluster: "yes"
```

In the `multi_packtpub_prod` inventory file, assign the Masakari services to run the API and engine services in the cloud controller nodes:

```
...
[masakari-api:children]
control
[masakari-engine:children]
control
```

At the cloud controller level, the instance evacuation monitor requires that the compute nodes that will deploy Peacemaker and Corosync under the hood to be monitored:

```
[masakari-hostmonitor:children]
control
```

Here, `hacluster` is an extra Ansible role that provides support to Masakari for compute node failure monitoring and recovering instances on a healthy hypervisor node, as required for Pacemaker and Corosync:

```
[hacluster:children]
control
[hacluster-remote:children]
compute
```

The instance restart monitor requires the instance status to be monitored by the compute nodes themselves:

```
[masakari-instancemonitor:children]
compute
```

Run the pipeline and make sure the Masakari containers are up and running by checking the new Masakari containers. To do so, run the following command line in a cloud controller node:

```
$ sudo docker ps
```

We will get the following output:

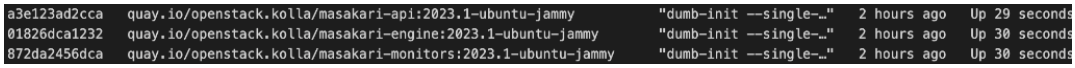


Figure 7.22 – Listing Masakari Kolla containers

To simulate a VM HA scenario, we will spawn an instance in one compute node and use the Masakari instance restart monitor mode:

```
$ openstack server create --image cirros-5.1 --flavor m1.tiny
--network priv_net vm-ha
```

Set the `HA_Enabled` flag property of the created instance to `True`:

```
$ openstack server set --property HA_Enabled=True vm-ha
```

Check which compute node the instance is running and kill the instance process ID to simulate an instance failure:

```
$ openstack server show vm-ha
```

Here is the output:

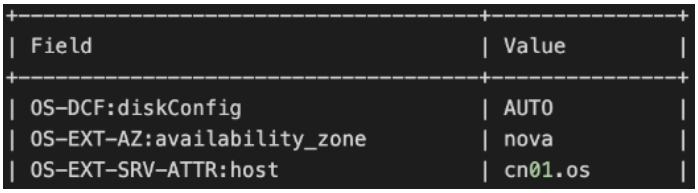


Figure 7.23 – Listing the virtual machines for the HA test

Log in to the compute node and kill the instance PID:

```
$ pgrep -f guest=instance-00000003
12652
$ kill -9 12652
```

By watching the Masakari log file outputs in `/var/log/kolla/masakari`, observe the status of the instance reboot in the same hypervisor host:

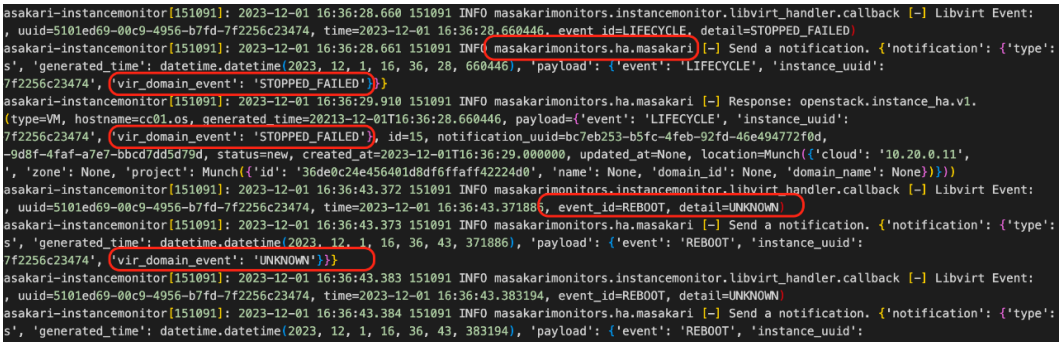


Figure 7.24 – Validating the Masakari instance respawning process in the log entries

Within a few seconds, the killed instance will be spawned in the same compute node. This can be checked via the newly created instance PID:

```
$ pgrep -f guest=instance-00000003
13442
```

As we can see, Masakari can be a great addition to ensure workload fault tolerance without the need to operate the HA capability separately for each instance, thus avoiding heavy manual effort. Masakari is getting more popular, and several production deployments have proven its advantages in helping workload administrators and cloud developers tackle failover challenges at the instance level.

Summary

In this chapter, we enabled a critical architectural safeguard design in our OpenStack environment by covering the HA pillar across the control and data planes. We now have numerous ways to construct a highly available OpenStack environment, depending on which HA strategy is preferred. In this chapter, we empowered the control plane services using HAProxy and Keepalived. Other deployments could employ Corosync, Pacemaker, and a selection of vendor solutions for load balancing. As demonstrated in this chapter, more than a single design pattern can be applied for common infrastructure services, such as databases and message queues, to achieve failover at best. Networking HA in OpenStack has achieved another level of maturity that enables cloud operators to choose between two options to minimize any potential risks of connectivity loss for the tenant networks: routing with VRRP and DVR.

This chapter also explored an attractive capability for cloud users to achieve failover with native OpenStack services for their workloads running in instances using Masakari. We also showed some additional snippets for managing HA in OpenStack layers using `kolla-ansible`. There is still some

ongoing amelioration on the infrastructure code using Kolla to automate cell and region deployments for compute services in OpenStack that was not covered in this chapter. HA and redundancy are critical elements that are required for you to start welcoming production workloads in your OpenStack environment and with this chapter, you should be ready to get started. Alongside ensuring the HA of services, we need to keep an eye on them and act proactively when things start to boil.

The next chapter will discuss the next operational excellence pillar of the OpenStack journey. It will explore ways you can monitor your cloud environment for the early detection of anomalies and deep dive into more fine-grained options for introspection and the analysis of collected OpenStack services logs.

8

Monitoring and Logging – Remediating Proactively

"To accept something on mere presumption and, likewise, to fail to investigate it may cover over, blind, and lead astray."

– Abu Nasr Al Farabi

Monitoring the underlying infrastructure of the private cloud is an essential operational activity that requires consistent tools and automated ways to achieve real-time metrics collection and alerting mechanisms. Among the different OpenStack releases, a variety of open-source and commercial tools exist to support the monitoring of different pieces of the OpenStack ecosystem. In the latest releases, we can find more emerging tools with successful monitoring stories, offering more simplicity and easier integration into the OpenStack ecosystem. Collecting metrics data is one part of OpenStack's operational tasks; the other part is the log data. Each OpenStack and shared service generates a load of log data with different logging levels, which, combined, would become a complicated task to manually parse and find anomalies when issues occur. Having a robust logging pipeline is a requirement to satisfy one pillar of the OpenStack operational journey. This chapter will combine both monitoring and logging practices with recent emerging tools that can be integrated into a running OpenStack environment.

We will discuss the following topics:

- Introducing Prometheus as a solution for the OpenStack infrastructure and workload monitoring
- Deploying and integrating Prometheus in the current OpenStack deployment and starting to use a separate monitoring stack to expose OpenStack metrics
- Exploring, deploying, and building dashboards using Grafana as a central observability tool for data representation
- Building an efficient and centralized logging solution using OpenSearch to ship, parse, and build custom search queries for collected logs, enabling further analysis

OpenStack infrastructure monitoring

Beyond the workloads running in OpenStack tenants, cloud operators must ensure that they have the necessary metrics to ensure OpenStack services' availability, performance, and usage status. Monitoring data should involve mainly two different categories of the OpenStack infrastructure services, as follows:

- **Cloud software**: This includes OpenStack APIs, schedulers, databases, message busses, and load-balancing services.
- **Under cloud**: This includes running operating systems, storage, and network hardware.

In the following section, we will explore Prometheus, a well-integrated and adopted monitoring and alerting toolkit for large-scale deployment.

Prometheus in a nutshell

Prometheus allows you to collect and store metrics as time-series data. Prometheus is considered a great fit for large deployment, as it provides exporters with components that ship various amounts of metrics to a Prometheus server. Additionally, the tool includes an alerting mechanism to manage alerts and notifications based on configured thresholds. Unlike other tools, such as Zabbix or Nagios, Prometheus offers flexible ways to feed metrics and data collected in its format. This is performed by third-party exporters that provide Prometheus-compatible metrics with less effort to develop separate converters for each system of the software framework. Some of the official exporters are maintained in Prometheus's GitHub repository and can be found at `https://github.com/prometheus`. In the OpenStack world, a vast number of metrics can be exported to Prometheus via the OpenStack Prometheus exporter, with a full list at `https://github.com/openstack-exporter/openstack-exporter`. As shown in the following diagram, Prometheus also has a discovery service that can be used by the OpenStack APIs to explore and generate data metrics for the Prometheus server. This is mainly useful for reporting metrics for virtual machines – for example, when utilizing the service discovery on the Nova API service. As soon as metrics are collected and stored by Prometheus, a single pane of glass should exist to consolidate and organize all data in comprehensible dashboards. Grafana is a great open-source tool for metrics visualization that can be configured to use Prometheus as its data source.

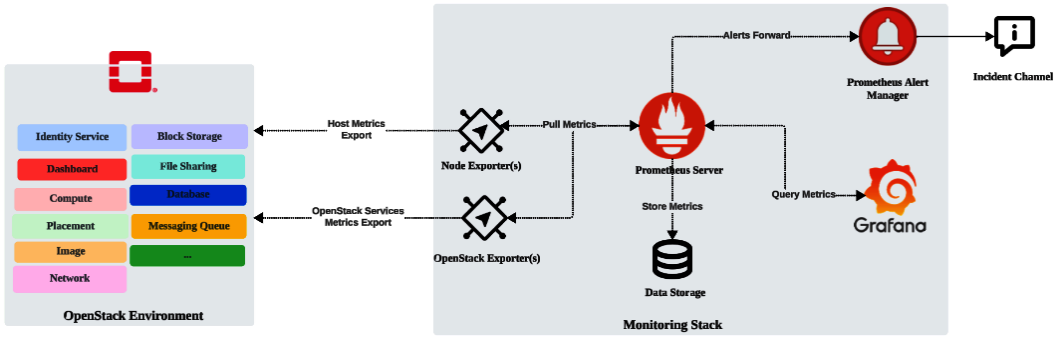


Figure 8.1 – A Prometheus integration and monitoring stack in OpenStack

In the following section, Prometheus will be deployed and integrated as the main monitoring tool in the OpenStack environment.

Deploying infrastructure monitoring

Adopting the Prometheus and Grafana pair as a central monitoring and observability solution for the OpenStack infrastructure is a quick win for cloud operators, due to the well-tested and integrated monitoring solution in the latest OpenStack releases. Kolla Ansible promotes and supports Prometheus and Grafana integration in the latest code releases. An additional monitoring host can be dedicated to run the Prometheus server and, optionally, Grafana. This can be achieved through the following configuration in the infrastructure as code settings:

1. Update the `multi_packtpub_prod` inventory file by adding the new monitoring host, as follows:

```
...
[monitoring]
mon01.os.packtpub
```

2. Add the Prometheus service to the monitoring host group:

```
...
[prometheus:children]
monitoring
```

3. Add the built-in Prometheus alert manager service to the monitoring host group:

```
...
[prometheus-alertmanager:children]
monitoring
```

4. Include the OpenStack Prometheus exporter in the monitoring host group. This will create a dedicated service container to automatically pull formatted Prometheus data from the OpenStack services' metrics:

```
...
[prometheus-openstack-exporter:children]
monitoring
```

5. Add the Prometheus Node Exporter role to run in each host. The Node Exporter provides a wide range of Linux operating system and hardware metrics:

```
...
[prometheus-node-exporter:children]
monitoring
control
compute
network
storage
```

6. Prometheus provides a dedicated exporter for the **MariaDB database**. The MySQL server metrics exposed through the exporter provide an extended list of data metrics available for scraping. The Prometheus MySQL exporter can be added to the `mariadb` role running in the cloud controller nodes:

```
...
[prometheus-mysqld-exporter:children]
mariadb
```

7. Similarly, Prometheus can be configured to scrape the load balancing metrics and stats running HAProxy via its dedicated exporter running in the load balancer nodes:

```
...
[prometheus-haproxy-exporter:children]
loadbalancer
```

8. Another available exporter that exposes both host and instance metrics is the **Prometheus Libvirt exporter**. This exporter collects data from the **Libvirt API** and exposes useful metrics for compute nodes and running instances. The `libvirt` exporter can be installed in the compute nodes as follows:

```
...
[prometheus-libvirt-exporter:children]
compute
```

9. For a broader monitoring setup across the OpenStack infrastructure, consider keeping an eye on the status of the containers running in each node. The Prometheus `cAdvisor` (short for **Container Advisor**) exporter is designed to collect, analyze, aggregate, and export container performance and metrics data. The `cAdvisor` exporter will run across all OpenStack nodes as follows:

```
...
[prometheus-cadvisor:children]
monitoring
control
compute
network
storage
```

10. The next step is to enable and install the Prometheus service using `kolla-ansible`. At the time of writing this edition, `kolla-ansible` will install the latest version of the Prometheus server after 2.0 by adding the following variable to the `globals.yml` file:

```
...
enable_prometheus: "yes"
```

11. For data exposure and visualization, enable Grafana in the `globals.yml` file:

```
...
enable_grafana: "yes"
```

12. `kolla-ansible` allows custom settings for the monitoring layer for both the Prometheus and Grafana services. Once both services are installed, you will need to provide a username and password for access. By default, `kolla-ansible` installs Prometheus using basic authentication via HTTP, with an `admin` user and a Grafana user, `grafana`. Both default variables are assigned to the `/kolla-ansible/ansible/group_vars/all.yml` file as follows:

```
...
prometheus_alertmanager_user: "admin"
prometheus_grafana_user: "grafana"
```

For custom user access settings, you can override the default usernames by adjusting the variables directly from the `all.yml` file or in the `prometheus_basic_auth_users` configuration section in the `/kolla-ansible/ansible/roles/prometheus/defaults/main.yml` file, as follows:

```
...
prometheus_basic_auth_users_default:
  - username: admin
    password: "{{ prometheus_password }}"
```

```
    enabled: true
  - username: "{{ prometheus_grafana_user }}"
    password: "{{ prometheus_grafana_password }}"
    enabled: "{{ enable_grafana }}"
```

User passwords for both Prometheus and Grafana can be reconfigured in the `/etc/kolla/passwords.yml` file, where all service secrets are defined. Both passwords are defined with the following variables:

```
prometheus_password:
prometheus_grafana_password:
```

The other option is to create new users using either custom passwords or the default ones, without overriding the existing ones, by extending the `prometheus_basic_auth_users_extra` variable in the `/kolla-ansible/ansible/roles/prometheus/defaults/main.yml` file, as follows:

```
prometheus_basic_auth_users_extra:
   - username: prometheus_pp_admin2
     password: "{{ prometheus_password }}"
     enabled: true
   - username: prometheus_pp_2
     password: prometheus_pp_password
     enabled: true
```

13. Another great metrics-scraping way is to configure a custom Prometheus run by listing a set of startup parameters in the `prometheus_cmdline_extras` variable, in the `globals.yml` file. For example, Prometheus can be configured to retain metrics data up to a specific size, via the `--storage.tsdb.retention.size` parameter. In the following listing, a customized configuration is defined to limit the maximum number of simultaneous connections on the Prometheus dashboard to `30` connections, adjust the log severity level to include `error` (the default severity level is `info`), increase the frequency of resending an alert to the Prometheus alert manager every `30` seconds, and retain metrics data for `30` days in storage:

```
prometheus_cmdline_extras: " --web.max-connections 30 --log.
level error  --rules.alert.resend-delay 30s  --storage.tsdb.
retention.time 30d "
```

Important note

A full list of the Prometheus startup parameters can be found at `https://prometheus.io/docs/prometheus/latest/command-line/prometheus/`.

14. Run a pipeline to deploy the Prometheus server, Grafana, Prometheus cAdvisor, and relevant exporter components. Once the deployment pipeline is completed, verify the newly deployed containers by running the following command line in one of the controller nodes:

```
$ docker ps -a | grep -e prometheus -e grafana
```

We get the following output:

```
6dae3127daa2   quay.io/openstack.kolla/prometheus-openstack-exporter:2023.1-ubuntu-jammy   "dumb-init --single-…"   2 hours ago   Up
0f5bd3e873b0   quay.io/openstack.kolla/prometheus-alertmanager:2023.1-ubuntu-jammy         "dumb-init --single-…"   2 hours ago   Up
df2ea6725451   quay.io/openstack.kolla/prometheus-cadvisor:2023.1-ubuntu-jammy             "dumb-init --single-…"   2 hours ago   Up
8872ae3caba2   quay.io/openstack.kolla/prometheus-haproxy-exporter:2023.1-ubuntu-jammy     "dumb-init --single-…"   2 hours ago   Up
653e34adcabd   quay.io/openstack.kolla/prometheus-mysqld-exporter:2023.1-ubuntu-jammy      "dumb-init --single-…"   2 hours ago   Up
ee3276edf910   quay.io/openstack.kolla/prometheus-node-exporter:2023.1-ubuntu-jammy        "dumb-init --single-…"   2 hours ago   Up
23ead127da31   quay.io/openstack.kolla/prometheus-v2-server:2023.1-ubuntu-jammy           "dumb-init --single-…"   2 hours ago   Up
092dacba5e54   quay.io/openstack.kolla/grafana:2023.1-ubuntu-jammy                        "dumb-init --single-…"   2 hours ago   Up
```

Figure 8.2 – Listing the Prometheus and Grafana Kolla containers

In the following section, we will check out the Prometheus-collected metrics and build insightful dashboards using Grafana.

Exploring the monitoring data

The latest `kolla-ansible` deployment enables the Prometheus server to start collecting different metrics available for scraping that can be checked on the Prometheus dashboard. The main Prometheus dashboard can be accessed by pointing to the IP address of the monitoring host: `http://host_monitoring_ip/prometheus`, where `host_monitoring_ip` is the IP address of the monitoring host running the Prometheus server.

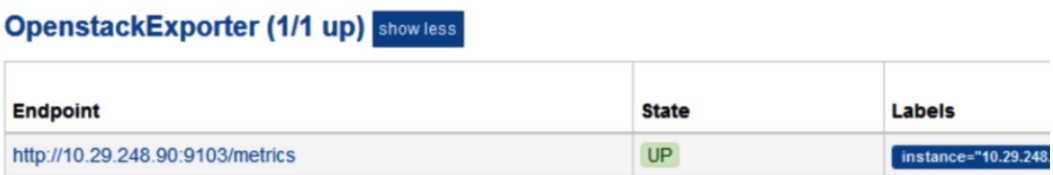


Figure 8.3 – The Prometheus dashboard interface

Prometheus collects an exhaustive list of metrics from each installed exporter across all defined hosts in the inventory. It uses **labels** to define each metric's target endpoint. For example, once the OpenStack exporter is installed, Prometheus starts collecting OpenStack-related metrics that can be checked by pointing to `http://host_monitoring_ip:9090/targets`, where `host_monitoring_ip` is the IP address of the monitoring host running the Prometheus server.

Collected metrics will be visible in the Prometheus dashboard as soon as the exporter service in each node sends metrics data for scraping. It might take a while to visualize all the metrics in the Prometheus dashboard. For example, the following listing shows that the initial OpenStack service exported metrics through `http://host_monitoring_ip:9100/metrics`, where `host_monitoring_ip` is the IP address of the monitoring host running the Prometheus server:

```
# HELP openstack_exporter_cache_refresh_duration_seconds Cache refresh d
# TYPE openstack_exporter_cache_refresh_duration_seconds gauge
openstack_exporter_cache_refresh_duration_seconds{region="RegionOne"} 0.
# HELP openstack_check_glance_api Openstack API check. fail = 0, ok = 1
# TYPE openstack_check_glance_api gauge
openstack_check_glance_api{region="RegionOne",service="glance",url="http
# HELP openstack_check_nova_api Openstack API check. fail = 0, ok = 1 an
# TYPE openstack_check_nova_api gauge
openstack_check_nova_api{region="RegionOne",service="nova",url="http://1
# HELP openstack_check_neutron_api Openstack API check. fail = 0, ok = 1
# TYPE openstack_check_neutron_api gauge
openstack_check_neutron_api{region="RegionOne",service="neutron",url="ht
# HELP openstack_check_placement_api Openstack API check. fail = 0, ok =
# TYPE openstack_check_placement_api gauge
openstack_check_placement_api{region="RegionOne",service="placement",url
# HELP openstack_check_keystone_api Openstack API check. fail = 0, ok =
# TYPE openstack_check_keystone_api gauge
openstack_check_keystone_api{region="RegionOne",service="keystone",url="
# HELP openstack_services_neutron_dhcp_agent_disabled_total Openstack Ne
# TYPE openstack_services_neutron_dhcp_agent_disabled_total gauge
```

Figure 8.4 – A listing of the collected OpenStack service metrics in Prometheus

The Node Exporter service installed on each OpenStack physical machine to gather host metrics, such as memory usage and stats, can be checked through the metrics listing at the same Prometheus dashboard URL, as shown in the following example:

```
go_memstats_last_gc_time_seconds 1.5511015252871068e+09
# HELP go_memstats_lookups_total Total number of pointer lookups.
# TYPE go_memstats_lookups_total counter
go_memstats_lookups_total 0
# HELP go_memstats_mallocs_total Total number of mallocs.
# TYPE go_memstats_mallocs_total counter
go_memstats_mallocs_total 6.93564679e+08
# HELP go_memstats_mcache_inuse_bytes Number of bytes in use by mcache structures.
# TYPE go_memstats_mcache_inuse_bytes gauge
go_memstats_mcache_inuse_bytes 13824
```

Figure 8.5 – A listing of the collected memory metrics and stats in Prometheus

With more nodes joining the OpenStack infrastructure, the number of collected metrics could be doubled, which would affect the monitoring host performance and disk capacity. For this reason, it is highly encouraged to use more than one Prometheus server instance to handle more metrics export and scraping jobs. Prometheus is designed to scale well in large deployments, with gazillions of resource types and metrics.

Browsing hundreds of metrics on the Prometheus dashboard to follow up on hundreds of service statuses and related usage information can be cumbersome. Prometheus can be very helpful with most essential operational tasks, including the following:

- Metrics data collection configuration
- Management and configuration of alerts
- Troubleshooting metrics export issues

Prometheus is a highly configurable tool and provides a native command-line interface to customize metrics collection and scraping settings, which are beyond the scope of this book. We will use the exported metrics data by Prometheus and visualize it in a simple Grafana dashboard.

The latest `kolla-ansible` run should enable the Grafana service with the default dashboard and configure Prometheus as a data source. To validate the Prometheus data source, point to `http://host_monitoring_ip:3000`, where `host_monitoring_ip` is the IP address of the monitoring host running the Grafana instance. Provide the username and password to access the Grafana dashboard configured in the previous steps. If you did not customize the Grafana access credentials, use the username and password pair as `admin/admin`, respectively.

Once logged in, navigate in the dashboard to the settings icon | **Configuration** | **Data sources**, and make sure that the Prometheus data source is added. Verify that the data source is functional, and no errors are reported. Click on the **Save & test** button, which should display that the data source configuration is functional, as shown in the following screenshot:

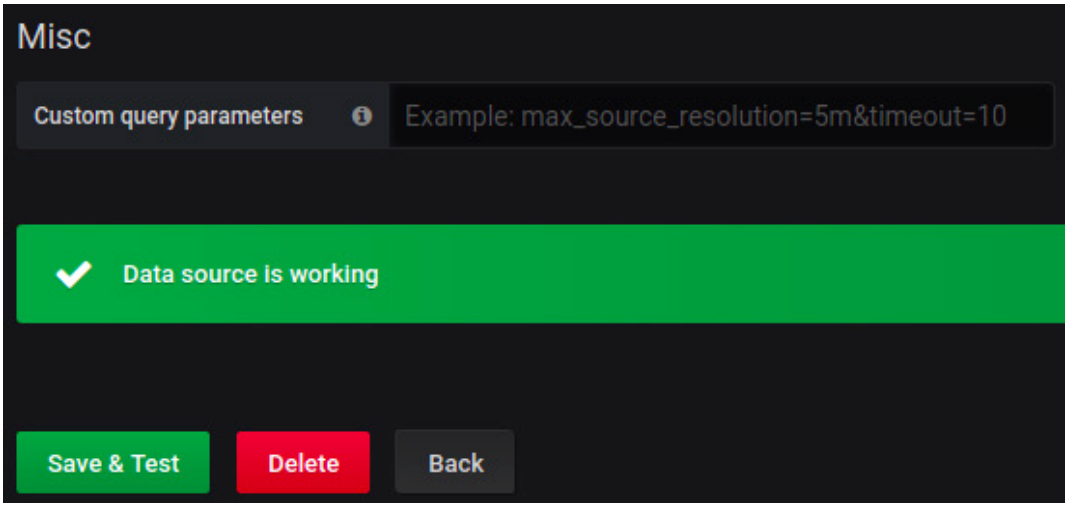


Figure 8.6 – Adding a Prometheus data source in the Grafana dashboard

Once the data source is checked, building dashboards in Grafana is straightforward and can be done either by creating freestyle dashboards or by importing dashboards from the Grafana website catalog. `https://grafana.com/` exposes different dashboards for each Prometheus exporter, identified by a dashboard ID that can be simply added to the local Grafana server. You can also import the dashboard in the JSON format and copy and paste it into the Grafana import section. In the next wizard, we will import Grafana dashboards for the OpenStack and Node Exporters using the **dashboard import IDs** option. To do so, simply navigate to the Grafana home page, then click on the **+** icon and select **Import**:

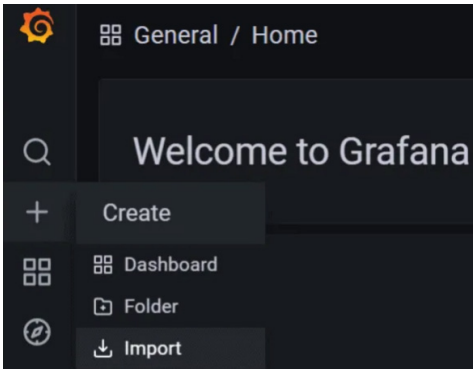


Figure 8.7 – Adding a dashboard in Grafana

Starting with the OpenStack metrics exporter, navigate to the OpenStack Grafana dashboard at `https://grafana.com/grafana/dashboards/9701-openstack-dashboard/` and copy the ID of the dashboard, as follows:

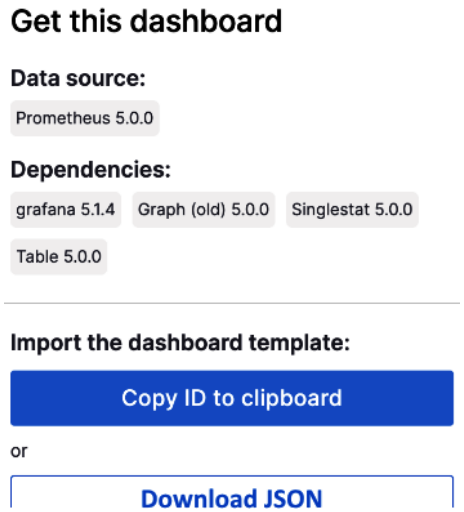


Figure 8.8 – Obtaining the dashboard ID

Paste the copied dashboard ID into the Grafana dashboard **Import** page. The dashboard ID should be `9701`:

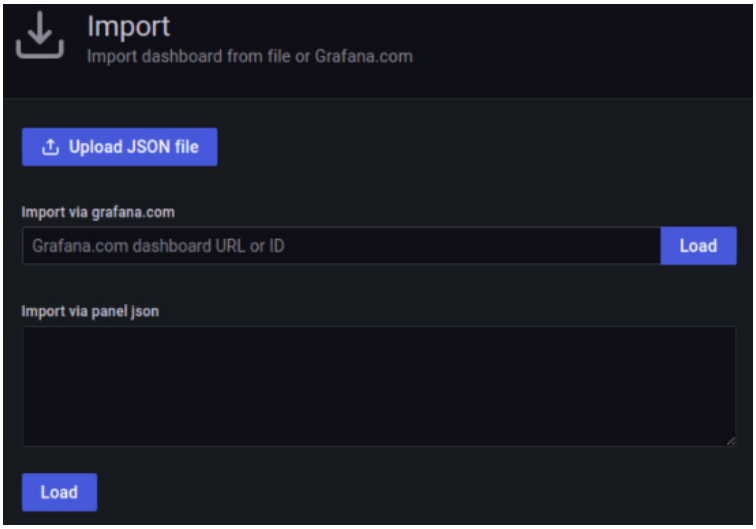


Figure 8.9 – Importing the dashboard ID into Grafana

The next page of the import wizard will guide you through the settings of the imported dashboard. Enter the name of the dashboard of your choice, and make sure that the **Prometheus** data source is selected, as shown in the following screenshot:

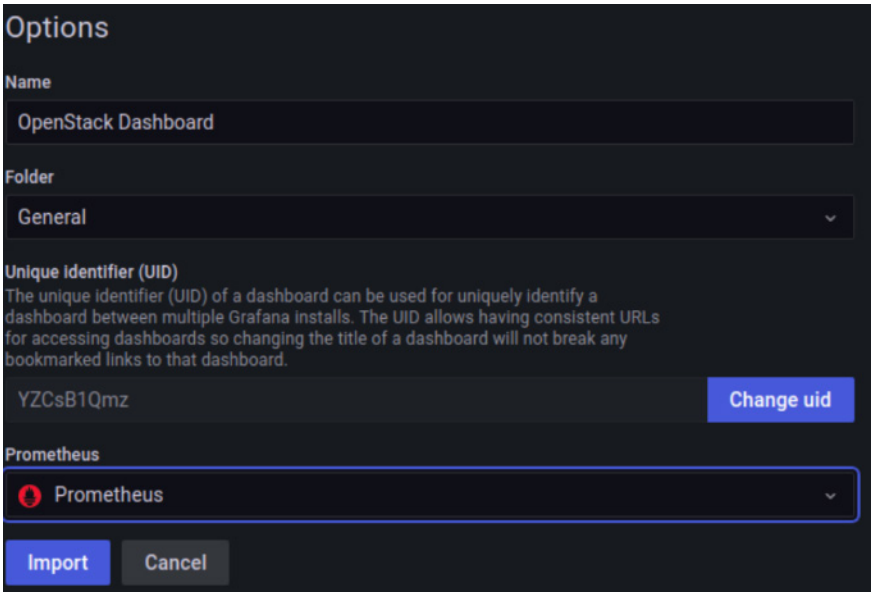


Figure 8.10 – Configuring the dashboard with Prometheus as the data source

Once the import wizard has completed, the imported Grafana dashboard for the OpenStack exporter metrics will display a variety of graphs and metrics, such as a generic overview of a resource's usage per OpenStack service. Some highlights of the services metrics are as follows:

- The current existing Keystone users, groups, and projects
- Neutron resources such as the floating IPs, virtual networks, and security groups deployed
- The number of running instances by the Nova service
- The current count of Glance images
- The volumes and snapshots managed by the Cinder service
- The overall usage of the CPU, RAM, and disk resources

Several dashboards can be displayed, as shown in the following screenshot:



Figure 8.11 – The imported generic dashboard with Prometheus as the data source

The imported dashboard has more insights into each service and its relative subcomponents, such as API services and agents.

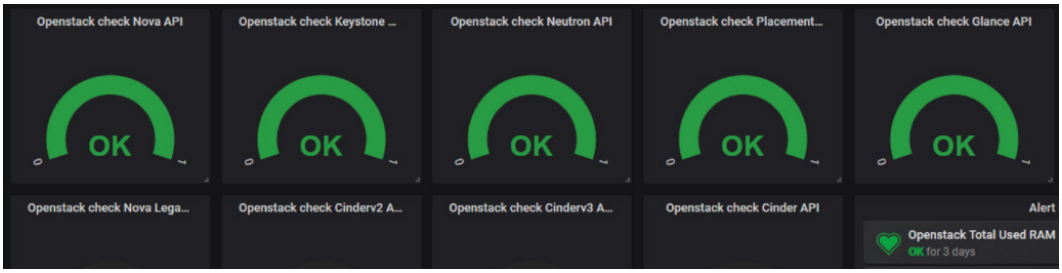


Figure 8.12 – The imported dashboards for each OpenStack API service status

Another useful dashboard should exist to monitor the hosts themselves running the cloud environment, via the Prometheus Node Exporter. The Grafana template dashboard can be found at `https://grafana.com/grafana/dashboards/1860-node-exporter-full/`, with the dashboard ID `1860`. Follow the same import instructions performed for the OpenStack exporter dashboard to make sure the data source selected is Prometheus.

Once imported, a new Grafana namespace will be created, with Prometheus as the data source. The collected data reflects granular and insightful information about the operating system and hardware metrics collected from each OpenStack host, using the installed Node Exporter service:

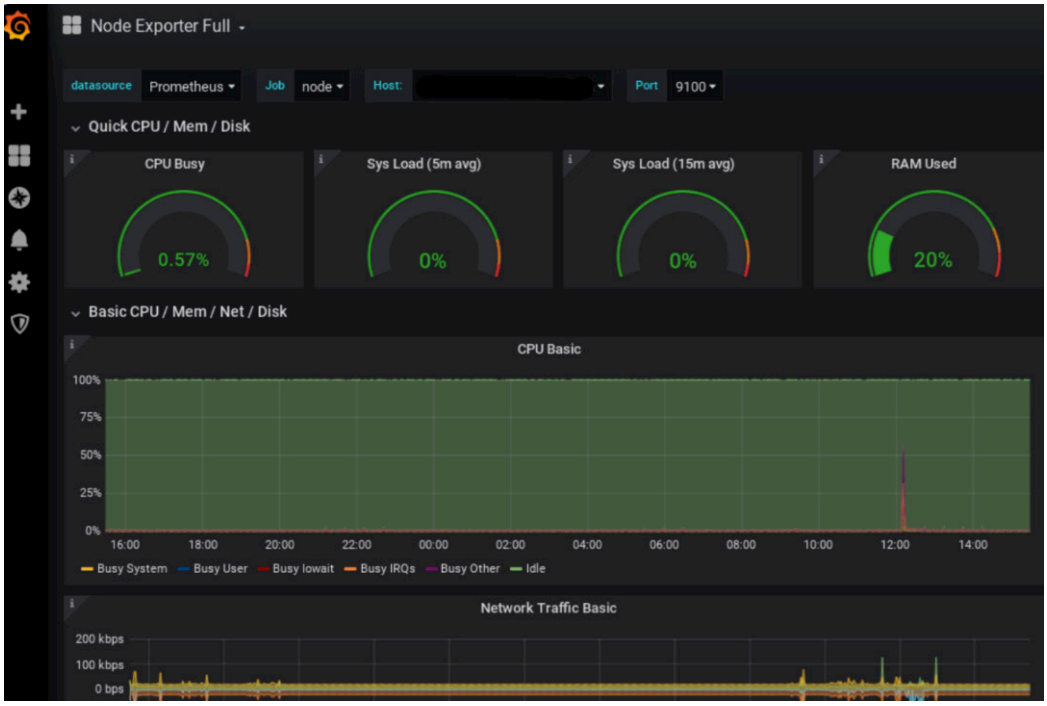


Figure 8.13 – A consolidated dashboard for OpenStack

Similarly, you can import ready-to-go Grafana dashboards for each installed Prometheus exporter. The database metrics dashboard can be found at `https://grafana.com/grafana/dashboards/14057-mysql/`, and HAProxy is available at `https://grafana.com/grafana/dashboards/2428-haproxy/`.

As far as data collected from the hosts running OpenStack services is concerned, the Prometheus Node Exporter service supports proactively identifying possible system anomalies. The imported dashboard consolidates all hosts' metrics in one single pane of glass that can be filtered by one or a group of hosts, offering a more custom visualization experience.

Next, we will move on to keep track of the workloads running on top of the OpenStack private cloud.

OpenStack workload monitoring

The next remaining part of the operational task of cloud operators is to provide monitoring support to the running workloads of the OpenStack tenants. Operators should also monitor some generic metrics of the provisioned resources by tenants to anticipate quota limitations and gain insightful analysis of the infrastructure usage. This can be achieved by using the same toolsets covered in the previous section. Workload monitoring focuses more on the resources deployed by each tenant – more precisely, the Nova instances. Prometheus is capable of querying the Nova API, and along with its labeling configuration settings, you should be able to filter and organize a list of thousands of instances into a set of scraped targets. In the previous section, the Prometheus Libvirt exporter was deployed in each compute node (see *Step 8* of the *Deploying infrastructure monitoring* section). Thus, there is no need to reconfigure any part of the infrastructure hosts. Instances metrics should be available in the Prometheus server, but identifying any reported instances with additional information is optional. Prometheus can be configured to discover instances in OpenStack by applying exporter configuration and the `relabel_configs` settings, as shown in the following example:

```
- job_name: 'openstack_pp'
openstack_sd_configs:
  - role: 'instance'
...
relabel_configs:
  - source_labels: [__meta_openstack_public_ip]
    target_label: __address__
    replacement: '$1:9100'
  - source_labels: [__meta_openstack_tag_prometheus]
    regex: true.*
    action: keep
- source_labels: [__meta_openstack_tag_node_exporter]
 regex: true.*
    action: keep
 - action: labelmap
   regex: __meta_openstack_(.+)
```

The tag-relabeling configuration helps to pull the metadata of discovered instances that matches a regular expression in Prometheus. In the previous example, the `__meta_openstack_(.+)` regular expression is applied and added for each metric received for all different source labels. This is useful to visualize tenant resources with more specific information, such as the instance name, IP address, and instance status, as shown in the following Prometheus server endpoint dashboard example:



Figure 8.14 – A listing of instance metrics in the Prometheus dashboard

`kolla-ansible`'s default Prometheus configuration does not include a full set of labels for each exporter type. The main configuration of Prometheus for all exporters can be found in the `kolla-ansible/ansible/roles/prometheus/templates/prometheus.yml.j2` file. One way to include a custom relabeling configuration is to edit the latter file and amend each exporter section settings. The recommended way is to create a new custom configuration file – for example, one named `custom.yml` under the new config directory, such as `/etc/kolla/config/prometheus/prometheus.yml.d/`, and apply the new additional settings by running the pipeline. `kolla-ansible` will merge the additional configurations in the running Prometheus server, but bear in mind that that will require a short restart of the server to load the new configuration.

Visualizing the instance metrics in Grafana would require only building corresponding dashboards. Similar to how observability is performed for the host metrics, the Node Exporter and Libvirt template dashboards combine instance-specific metrics to build custom tabs for instances' resource usage per compute node. This includes an overview of the total resource consumption, such as the CPU, RAM, and allocated volumes. At the time of writing this edition, importing Grafana dashboards for virtual machine metrics using the Prometheus discovery service is not available.

A Grafana dashboard for instance metrics from `grafana.com` can be found at `https://grafana.com/grafana/dashboards/15330-openstack-instance-metrics/` with the dashboard ID `15330`. The dashboard cannot be used without a fully running **telemetry OpenStack** service (composed of **Ceilometer** and **Gnocchi**), which will be covered in the next section.

Telemetry and charging back

OpenStack provides telemetry for tenant environments with the Ceilometer service. The Ceilometer project aims to support cloud operators to collect metrics, further assisting and understanding resource utilization. This enables us to address infrastructure scalability issues and update resource planning for expansion and growth before hitting the resource limits. Since the Queens OpenStack

release, the telemetry module has been extended with additional components that work in tandem with Ceilometer, including the following:

- **Aodh**: An alarming service that acts upon collected metrics data
- **Gnocchi**: A data store and collection of time series data
- **Panko**: A data store for events data generated by Ceilometer

> **Important note**
> At the time of writing this edition, the **Panko** module has been deprecated and is no longer maintained by the OpenStack community.

The next section will highlight the telemetry stack in more detail.

The Ceilometer anatomy

The Ceilometer project originally aimed (until the Folsom release) to collect resources' usage metrics from cloud tenants and transform them into billable items for invoicing purposes. The continuous development of the Ceilometer project has enabled additional monitoring and alarming capabilities. They have been split into their own modules since the Liberty release.

As shown in the following diagram, Ceilometer relies on agents to collect, process, store, and retrieve data.

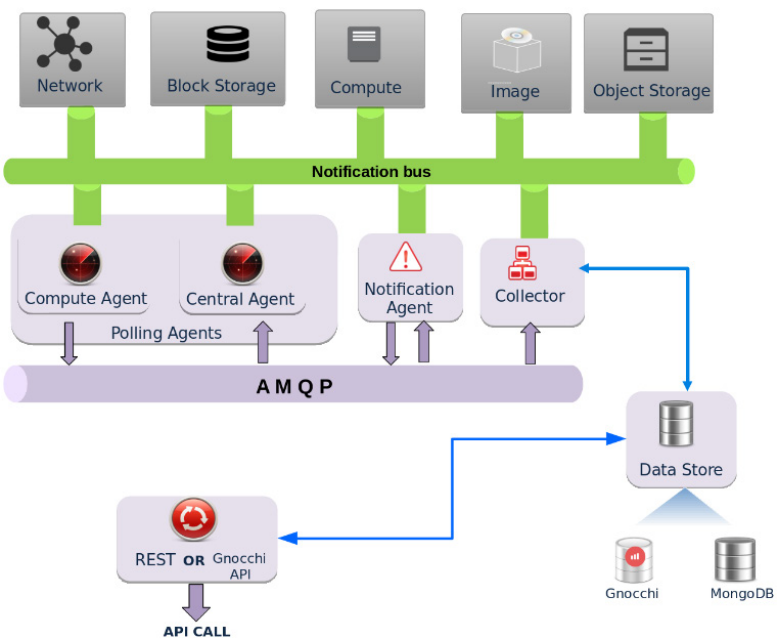


Figure 8.15 – A general overview of the Ceilometer architecture

Each data workflow is actioned by a dedicated agent that can be listed as follows:

- **Polling agents**: These regularly poll each OpenStack infrastructure service to formulate measurements via API calls. A compute polling agent gathers statistics from instances running in a compute node and polls them to the message queue. The central agent runs in a central management server in OpenStack, such as a cloud controller. It polls statistics of resources other than instances.
- **Notification agents**: These listen periodically on the message queue bus, collect new notification messages set by different OpenStack services, and translate them into metrics before pushing them back to the appropriate message queue bus.
- **Collector agents**: These monitor the message queue, gather samples, and collect the metering messages generated by either polling or notification agents. Therefore, the new metering messages will be recorded in a backend storage.
- **API service**: This exposes a standard API that provides access to the internal Ceilometer database, if enabled, to query metering data against it.

Gathering data is mainly performed by agents using a pipeline mechanism. Internally, they periodically send requests for sample objects that reflect a certain meter. Every sample request will be forwarded to the pipeline. Once passed to the pipeline, meters can be manipulated by several transformer types:

- **Accumulator**: This accumulates multi-values and sends them in a batch.
- **Aggregator**: This aggregates multi-values into one arithmetic. This includes arithmetic functions to compute the percentage.
- **Rate of change**: This identifies trends by deriving another meter from the previous data.
- **Unit conversion**: This gives the type of unit conversion to be used.

Once manipulated and transformed, a meter might follow its path via one of the multiple publisher types:

- **Notifier**: This is the meter data pushed over a reliable messaging queue.
- **rpc**: This is the synchronous RPC meter data publisher.
- **udp**: This is the meter data sent over the UDP.
- **file**: This is the meter data sent into a file.

The aforementioned components are illustrated as follows:

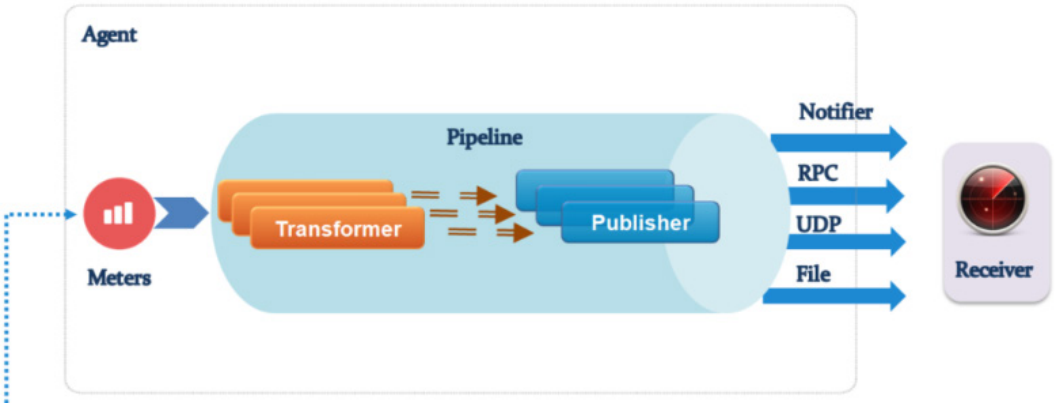


Figure 8.16 – Metrics data transformation and publishing through metric pipelines

Data collected can be stored in different types of databases supported by Ceilometer. Originally, MongoDB was a widely adopted data store for the telemetry metric storage. Due to MongoDB's scalability and performance issues, Gnocchi has been evaluated as a well-suited time series database for Ceilometer, which will be covered in the next section.

The Gnocchi data store

Since the Train OpenStack release, **Gnocchi** has been considered the default and official supported storage service for Ceilometer. Given the performance and scalability issues of former metrics data store options, the introduction of Gnocchi as a time series database has solved main bottlenecks issues when processing and storing metrics data in large-scale environments. Data samples are no longer written directly into a database. Rather, they are converted into Gnocchi elements and posted afterward on its native API. Aggregated data is recorded in a time series. Each converted sample presents a data point that has a timestamp and measurement.

The hallmark of data optimization using Gnocchi is indexing resources and their associated attributes. As a result, data searches are quicker. As mentioned previously, Gnocchi offers a scalable metric storage design by utilizing metric data storage in a scalable storage system, such as **Swift** and **Ceph**. As shown in the next figure, Gnocchi can be configured to aggregate and store measures in Swift or Ceph as a storage backend, using its built-in storage drivers:

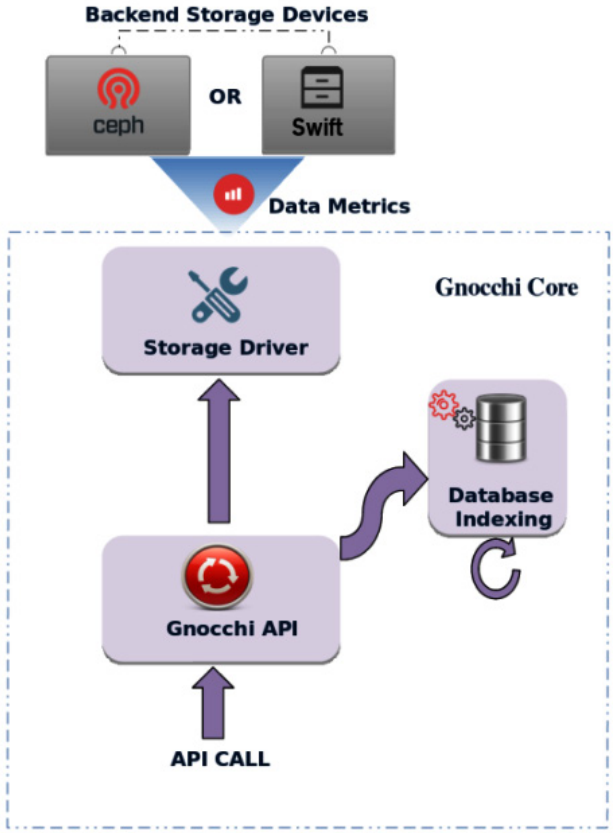


Figure 8.17 – The Gnocchi data store multi-backend support

Data is processed by `metricd` for aggregation and storage. `metricd` handles flushing metrics marked for deletion. Gnocchi provides a compatible `statsd` protocol (`gnocchi-statsd`) that listens to the incoming metrics. `statsd` will be used later for Gnocchi deployment.

Alerting with Aodh

Since the Liberty release, the Ceilometer alarm module has been forked and renamed to **Aodh**, which takes the lead in triggering alarms based on custom rules.

As shown in the following diagram, a significant difference can be noticed when comparing the current alarm service to Ceilometer's preceding one – the ability to scale horizontally when hitting more load. Using the same messaging queue, Aodh exposes an event listener that catches new notifications and provides an instant response time, with zero latency. Aodh listeners rely on predefined alarms that will trigger instantly based on events and configured measures. In this case, reacting against auto-scaling conditions when using an orchestration service (e.g., **Heat**) becomes very useful:

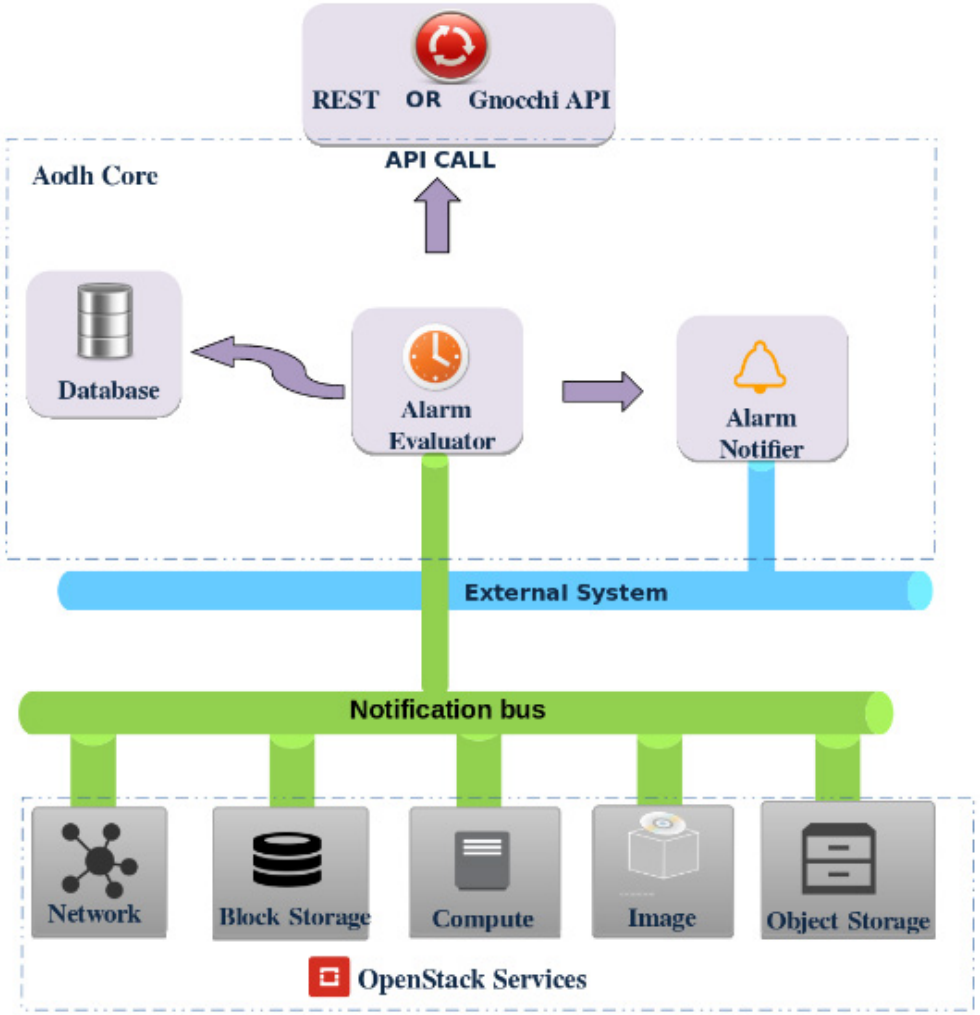


Figure 8.18 – An Aodh alarm overview in OpenStack

Aodh is constructed mainly by an API server (`aodh-api`) that provides access to the data store. The Aodh alarm evaluator (`aodh-evaluator`) triggers alarms, based on the collected statistical trends crossing a threshold within a certain period. Alarms are triggered by an Aodh notification listener (`aodh-listener`), based on rules against events reported by the notification agents. The Aodh alarm notifier (`aodh-notifier`) enables alarm settings based on the threshold for a collection of samples.

Aodh notification supports both event- and threshold-based alarms. In the latest OpenStack releases, Aodh queries measurements by default from the Gnocchi data store.

Deploying the telemetry service

The current `kolla-ansible` repository supports a full deployment of the telemetry service in OpenStack. This includes the Ceilometer, Aodh, and Gnocchi services. As we have already deployed the Ceilometer service in *Chapter 3*, *OpenStack Control Plane – Shared Services*, we will walk through a detailed configuration of the Ansible roles for the telemetry stack:

1. Make sure the Ceilometer service is added to run it as part of the cloud controller nodes if it is not enabled by adding it to the `multi_packtpub_prod` inventory file, as follows:

```
...
[ceilometer:children]
control
```

2. Add the Ceilometer central polling and notification agents to the cloud controller group:

```
...
[ceilometer-central:children]
ceilometer
[ceilometer-notification:children]
ceilometer
```

3. Add the ceilometer polling agent for the compute nodes group:

```
...
[ceilometer-compute:children]
compute
```

4. Run the Gnocchi service as part of the cloud controller nodes group:

```
...
[gnocchi:children]
control
```

5. Assign the Gnocchi components, including the API, `statsd` (the `gnocchi-statsd` daemon), and `metricd` (the `gnocchi-metricd` daemon) services, to run as part of the Gnocchi instance in the cloud controller nodes group:

```
...
[gnocchi-api:children]
gnocchi
[gnocchi-statsd:children]
gnocchi
[gnocchi-metricd:children]
gnocchi
```

6. Run the Aodh alarm service as part of the cloud controller nodes group:

```
...
[aodh:children]
control
```

7. Assign the Aodh components, including the API, evaluator, listener, and `notifier` services, to run as part of the Aodh instance in the cloud controller nodes group:

```
...
[aodh-api:children]
aodh
[aodh-evaluator:children]
aodh
[aodh-listener:children]
aodh
[aodh-notifier:children]
aodh
```

8. Next, make sure the Ceilometer service is enabled in the `globals.yml` file:

```
...
enable_ceilometer: "yes"
```

9. Enable the Gnocchi service in the `globals.yml` file and, optionally, the `statsd` protocol support that will install and run the `gnocchi-statd` daemon:

```
...
enable_gnocchi: "yes"
enable_gnocchi_statsd: "yes"
```

10. The last configuration in the `globals.yml` file is to enable the Aodh alerting service by setting its corresponding variable to `yes`:

```
...
enable_aodh: "yes"
```

11. Run the deployment pipeline and make sure that no errors have been reported when installing the telemetry service. Verify that the telemetry service containers are up and running in one of the cloud controller nodes using the following command line:

```
$ docker ps -a | grep -e ceilometer -e gnocchi -e aodh
```

This is the output we get:

```
d55aa3751b1d   quay.io/openstack.kolla/aodh-notifier:2023.1-ubuntu-jammy             "dumb-init --single-…"   2 hours ago   Up 2 hours (healthy)
b7f1ade8cf50   quay.io/openstack.kolla/aodh-listener:2023.1-ubuntu-jammy             "dumb-init --single-…"   2 hours ago   Up 2 hours (healthy)
5a2f03854727   quay.io/openstack.kolla/aodh-evaluator:2023.1-ubuntu-jammy            "dumb-init --single-…"   2 hours ago   Up 2 hours (healthy)
ca5054ffcb1c   quay.io/openstack.kolla/aodh-api:2023.1-ubuntu-jammy                  "dumb-init --single-…"   2 hours ago   Up 2 hours (healthy)
b3fe5de9c0d7   quay.io/openstack.kolla/ceilometer-central:2023.1-ubuntu-jammy        "dumb-init --single-…"   2 hours ago   Up 2 hours (healthy)
96e41a027685   quay.io/openstack.kolla/ceilometer-notification:2023.1-ubuntu-jammy   "dumb-init --single-…"   2 hours ago   Up 2 hours (healthy)
407a3b083aa9   quay.io/openstack.kolla/gnocchi-statsd:2023.1-ubuntu-jammy            "dumb-init --single-…"   2 hours ago   Up 2 hours (healthy)
8847afd527c6   quay.io/openstack.kolla/gnocchi-metricd:2023.1-ubuntu-jammy           "dumb-init --single-…"   2 hours ago   Up 2 hours (healthy)
6409e89ac7c9   quay.io/openstack.kolla/gnocchi-api:2023.1-ubuntu-jammy               "dumb-init --single-…"   2 hours ago   Up 2 hours (healthy)
```

Figure 8.19 – A listing of the deployed telemetry Kolla containers

Once the trio of Ceilometer, Gnocchi, and Aodh are joined to the OpenStack deployment, you should have a fully functional telemetry service. Navigating through Ceilometer metrics is straightforward and can be used to configure alarms via Aodh. Metrics and alarms are essential elements to tackle daily monitoring operations. However, even these would not be sufficient without the addition of service logs, which we will cover in the next section.

Grasping centralized logging

The second essential part of operating an OpenStack cloud environment is to embrace the inspection of events in the infrastructure through logging information. With dozens of deployed services involved in the OpenStack setup, cloud operators should enable logging in each service, running at least within the control plane for troubleshooting, analysis, and even creating custom metrics for alerting. Depending on the OpenStack deployment tool and configuration settings used, the standard services in Linux/Unix systems write their logs in the `/var/log/` directory and subdirectories.

With the tons of log data generated, parsing and processing it all presents a hindrance to extracting meaningful information or troubleshooting unexpected issues, which take a longer time to solve. To overcome such a challenge, your log environment must evolve to become centralized. A good option is to start flowing logs in a dedicated `rsyslog` server. You might put in so much data that your log server starts starving for larger storage capacity. Furthermore, archiving the aforementioned data will not be useful when you need to extract information for a particular context. Additionally, correlating log data that has different formats (taking into consideration the RabbitMQ and MySQL logs) with generated events might even be impossible. So, what we need at this point is a set of quality requirements for a good OpenStack logging experience, as follows:

- A parsing capability for log metadata
- Fast indexing of log data
- A comprehensible presentation of data
- Metrics constructed from custom log data aggregations

Generated logs in OpenStack are structured, making them suitable for processing, and can be parsed by several tools, including the open-source **ELK** stack, which is composed of **Elasticsearch**, **Logstash**, and **Kibana**, or even by commercial third-party tools, such as Datadog and Splunk. ELK is the most adopted tool as a logging pipeline in large OpenStack deployments. Most OpenStack deployment tools, other than `kolla-ansible`, are supported by an ELK stack installation out of the box. Although the ELK stack is still a valid option to manage logs in OpenStack, the latest OpenStack releases have shifted gears to adopt a forked open-source Elasticsearch solution, and **OpenSearch** was announced as the new way to handle logging in the next OpenStack operations.

OpenSearch under the hood

OpenSearch is an open-source software offered under the Apache version 2.0 license. It has become a very popular search engine and log analytics suite, due to its large community and active commits, and originated from the Elasticsearch project. The OpenSearch suite includes a search engine, data store, and dashboards for visualizations, which originated from Kibana. The OpenSearch architecture is based on a distributed design, where different components can run in different cluster nodes. The following diagram shows an overview of a logging solution based on OpenSearch for OpenStack:

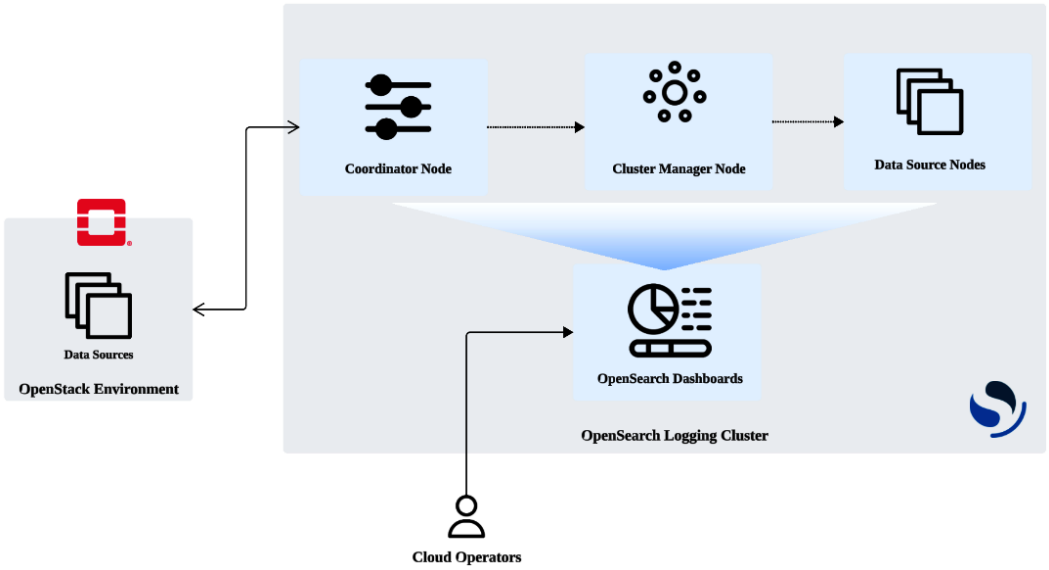


Figure 8.20 – An overview of the OpenSearch logging cluster in an OpenStack environment

As shown in the previous diagram, a typical OpenSearch environment for mid to large deployments consists of a variety of cluster node roles, such as the following:

- **The cluster manager node**: This keeps track of the OpenSearch formation cluster state, including node state changes (i.e., joiners and leavers) and health, indexing management, and shard allocation. For high availability reasoning, it is recommended to run at least two manager nodes for production use cases.
- **The coordinator node**: This receives requests initiated from a dashboard or client libraries and delegates them to the right shards on the data nodes, and then fetches, aggregates, and returns the data results to the client.
- **The data source node**: This represents the worker horses of the OpenSearch cluster that store the data and perform different data-related operations, including indexing, aggregation, and searching.

> **Important note**
>
> Other types, including the **master-eligible** and **ingest** nodes, can form part of an OpenSearch formation cluster. Any node that is not marked as a master is designated with a master-eligible role. Ingest nodes are recommended for heavy data ingestion pipelines and to offload indexing workloads from data nodes.

OpenSearch is highly extensible, and several types of plugins can be installed for an enhanced search experience, security features, machine learning, and performance analysis. The available plugins are listed at `https://opensearch.org/docs/latest/install-and-configure/plugins/#available-plugins`.

The OpenSearch solution comes with a distributed architecture, enabling you to scale for large search and analytics operations. Niche features are supported by the OpenSearch core search service, making it easy to run log analytics, data ingestion, and observability capabilities. Additionally, performance has become a valid reason to adopt OpenSearch, due to additional ways to handle shards across multiple nodes that speed up the search operation. The amount of data generated by different OpenStack and common infrastructure services can be immense and can be fed into an OpenSearch cluster formation. By providing a minimum size of an OpenSearch cluster, cloud operators can start exploring a rich search and visualization experience. This enables you to create visualizations to spot the malfunctioning parts of complex and large systems that are hard to detect with basic metrics.

Deploying OpenSearch

At the time of writing this edition, `kolla-ansible` supports OpenSearch installation. The community recommends the adoption of the OpenSearch option, which can be deployed in a few steps. It is highly recommended to dedicate a logging cluster that runs different OpenSearch node types, as covered in the previous section. Other considerations should be taken for each OpenSearch

node requirement so that you can tailor your logging cluster, based on the required performance. For example, data nodes would require more disk space and could be upgraded with fast disks with higher IOPS, such as **solid-state drives** (**SSDs**). Master and coordinating nodes are more demanding computers that can be deployed with higher CPU power. For the next setup, we will run a simple instance of the OpenSearch service as part of the control node:

1. Assign the OpenSearch engine and `dashboards` roles to the control node:

```
...
[opensearch:children]
control
[opensearch-dashboards:children]
opensearch
```

2. Change the following variable in the `globals.yml` file to install OpenSearch as a central logging service:

```
...
enable_central_logging: "yes"
```

3. By default, Kolla deploys the OpenSearch service to listen on port `9200`. When deployed in the controller nodes, OpenSearch Dashboards should be accessible via the reserved VIP assigned to Keepalived to run the cloud controller clusters. Optionally, you can assign an FQDN and populate the hostname internally across all OpenStack nodes. The dashboards can be accessed with a browser internally on port `5601` and the configured `kolla_internal_fqdn` variable set in the `globals.yml` file that points to the `kolla_internal_vip_address` variable. Make sure that at least the `kolla_internal_vip_address` variable is set to the VIP address or mapped DNS endpoint name:

```
kolla_internal_vip_address: "10.0.0.47"
```

4. OpenStack dashboards are created with the default `'opensearch'` username, and the password can be edited in the `kolla-ansible/etc/kolla/passwords.yml` file by setting the following variable:

```
opensearch_dashboards_password: 'openstestpass'
```

5. Other advanced and useful configurations that can be performed in advance to address common OpenSearch operation settings are the log retention policies, which are critical when dealing with disk-size management, data rotation, and the life cycle. OpenSearch comes with an **Index**

State Management plugin to manage data rotation through the definition of log retention policies. Make sure that is enabled in the Ansible OpenSearch role defined in the `ansible/roles/opensearch/defaults/main.yml` file by checking the following variable:

```
opensearch_apply_log_retention_policy: true
```

6. Once enabled, OpenSearch provides an indices data life cycle by defining the soft and hard retention periods. The soft retention period is defined by the `opensearch_soft_retention_period_days` variable in the `ansible/roles/opensearch/defaults/main.yml` file where indices are closed, no longer active, and staged for deletion, but they still occupy space on the disk and can be reopened. The hard retention period is the duration after which indices are deleted permanently and no longer occupy a space on the disk. They are defined by the `opensearch_hard_retention_period_days` variable in the `ansible/roles/opensearch/defaults/main.yml` file. The default soft and hard duration periods are set to `30` and `60`, respectively. That can be edited either by creating new variables in the `globals.yml` file and referencing them in the OpenSearch playbook, or directly in the `ansible/roles/opensearch/defaults/main.yml` file by changing the soft duration to `45` and the hard one to `90` days:

```
opensearch_soft_retention_period_days: "{{ elasticsearch_
curator_soft_retention_period_days | default(45) }}"
opensearch_hard_retention_period_days: "{{ elasticsearch_
curator_hard_retention_period_days | default(90) }}"
```

7. Run the pipeline and ensure that the output does not contain errors in the job configuration. After completion, run the following command line to validate the additional Kolla containers:

```
$ docker ps -a | grep -e opensearch
```

Here is the output:

```
d52da212711d   quay.io/openstack.kolla/opensearch:2023.1-ubuntu-jammy                "dumb-init --single-…"   1 hour ago   Up 1 hour (healthy)
c2bdae217621   quay.io/openstack.kolla/opensearch-dashboards:2023.1-ubuntu-jammy     "dumb-init --single-…"   1 hour ago   Up 1 hour (healthy)
```

Figure 8.21 – Listing of OpenSearch Kolla containers

8. Point to the OpenSearch Dashboards URL from a browser at `http://10.0.0.47:5601`. Enter the `'opensearch'` username and configured password to start a log search:



Figure 8.22 – The OpenSearch login page

9. To start searching logs after the first login, an index pattern must be created to pull data from the logs' sources. The index pattern points to a specific index that can pull log data from a specific date, for example. From the **OpenSearch Dashboards** panel, click on the **Index Patterns** button, as follows:

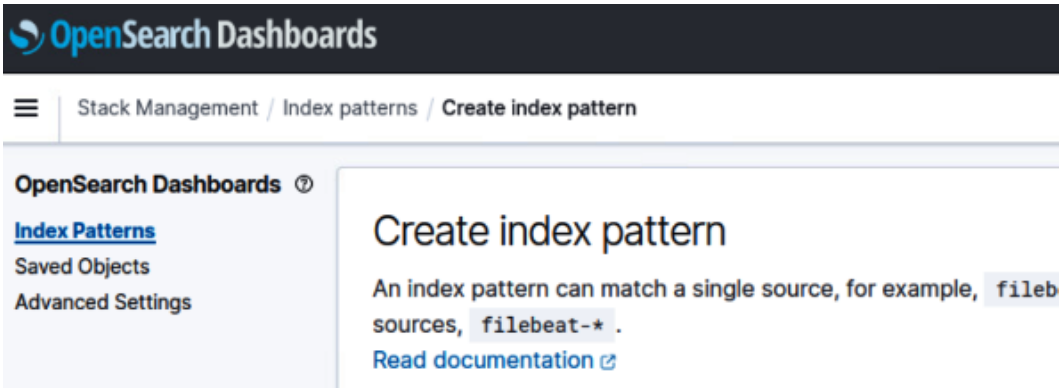


Figure 8.23 – Creating index patterns in the OpenSearch dashboard

10. Next, specify the index pattern settings by applying a filter to pull and aggregate data. In the current example, the **timestamp** filter will be applied. By default, OpenSearch associates a unique identifier to each index pattern, as follows:

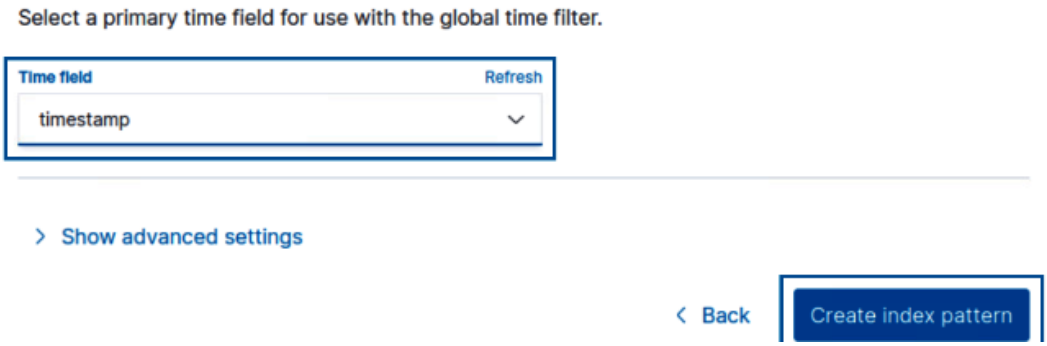


Figure 8.24 – Applying a filter for the index pattern

11. Once created, the index pattern will apply the filter to the source data and visualize the field's mapped core type, as saved by OpenSearch. For example, the created index pattern references a set of fields that are visible in the **Discover** tab on the upper side of the dashboard:

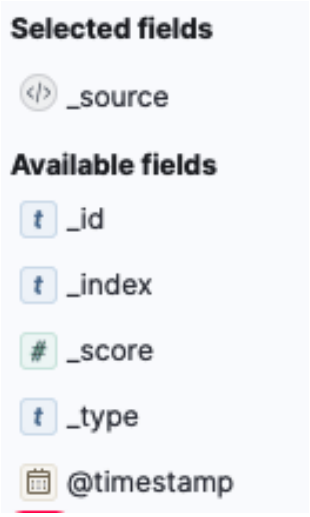


Figure 8.25 – Listing the field selection for the index pattern

12. Logs can be filtered using the search bar in the **Discover** panel, which lists all the associated events ending with a time frame, as shown in the following figure:

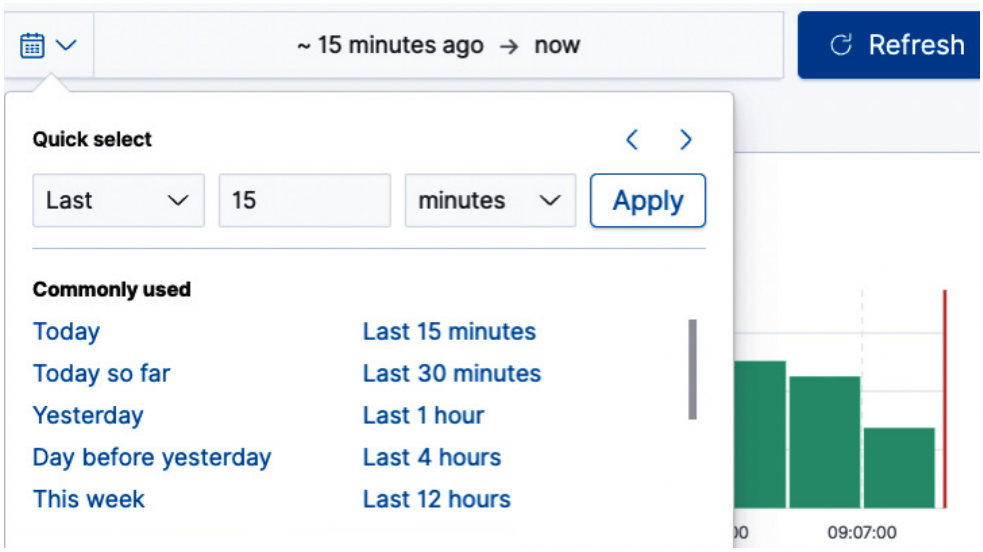


Figure 8.26 – Applying time frames for logs filtering

13. Further filtering can be applied using the search filters bar. In the following example, an added filter is applied to pull all data with an `ERROR` value in the `log_level` field:

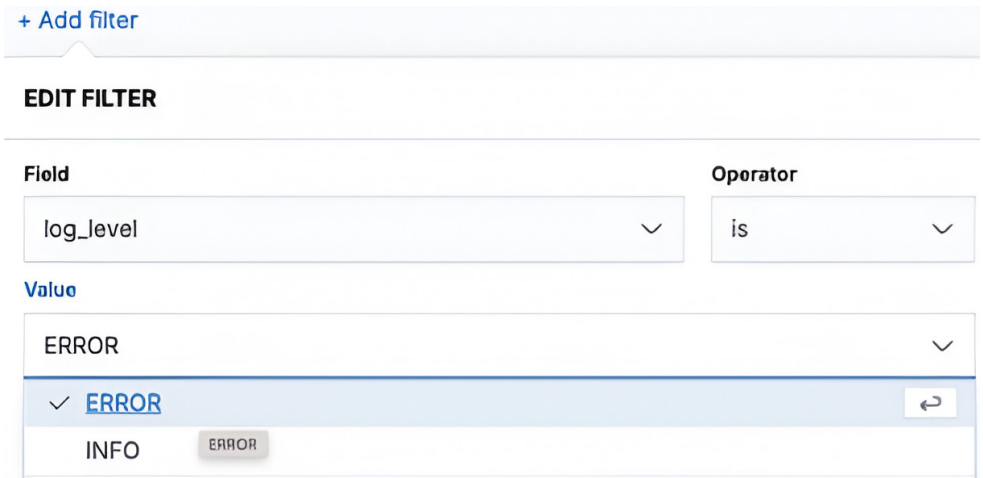


Figure 8.27 – Applying a log_level filter for log filtering

To quickly monitor the trend of the pulled data with a specific filter, it is important to use the power of OpenSearch's visualization capability. With a rich variety of supported dashboard objects, operators can build different types of visualizations for each search-filtered query. To create a visualization, point to the **Visualize** tab and click on the **Create Visualization** button. From the returned list, the **Vertical Bar Chart** visualization type is selected to expose the

collected error occurrences in the last hour from the last saved query search. Once created, a new graph is generated that can be broken up into more specific fields, referred to as **Buckets**. Data graph aggregation can be performed by adding, for example, a bucket in the created graph settings and selecting the **X-axis** option. As shown in the next example, the data aggregation is fixed with the **Date Histogram** type, with the default `@timestamp` field:

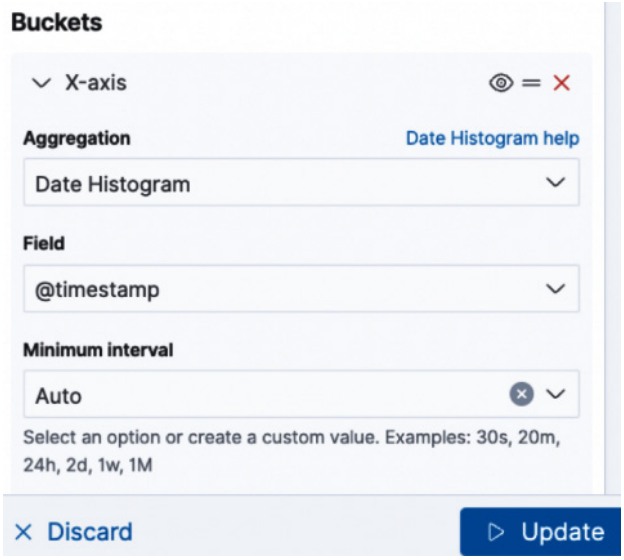


Figure 8.28 – Data aggregation using OpenSearch buckets

14. Once updated, a generic overview of the error rate across all pulled log data can be visualized in a single pane of glass:

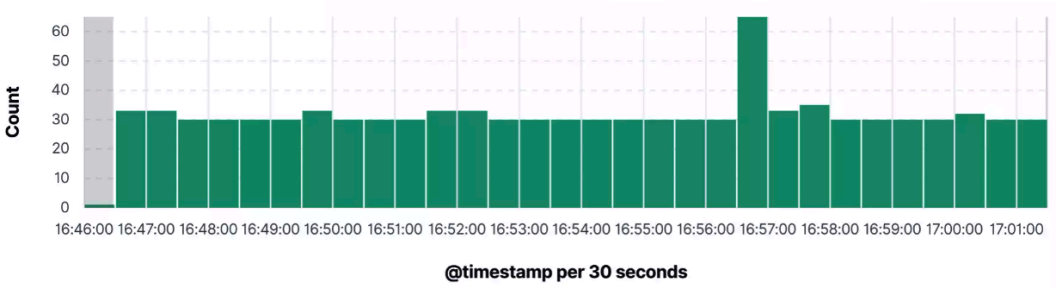


Figure 8.29 – The log visualization dashboard based on the applied filters

OpenSearch dashboards are highly customizable so that more data aggregation can be performed for more granular data presentation. For example, the previous graph representation can be saved in a dedicated service state dashboard, and then you can add a new visualization from the same graph by exposing the errors for each service.

Summary

This chapter covered two essential practices for operating an OpenStack environment, exploring the possible ways to monitor and log. Both deployed monitoring and logging stacks can vary from one setup to another, depending on the operator's experience and their familiarity with a specific tool, or some other third-party solutions. One of the best practices is to always monitor what is happening under and on top of the cloud platform, providing the necessary ways to capture metrics and collect data before it reaches a higher severity level and you have to start dealing with incidents. Prometheus, as demonstrated throughout this chapter, provides a multitude of configurations, and with the simple integration of exporters, cloud operators can easily ship metrics in an automated fashion. A single pane, Grafana, centralizes all the metrics in comprehensive dashboards, and operators can closely observe different system elements' trends. The OpenStack telemetry service, commonly referred to as the Ceilometer, Aodh, and Gnocchi trio, was covered and deployed to start charging and tracking resource usage for tenants. At the end of the chapter, the logging pipeline challenge was explored, as well as a widely adopted solution, OpenSearch, that enables the consolidation of the immense amount of generated logs in each corner of OpenStack, where you can explore suspicious events to remediate them proactively. Logs provide valuable information if combined in a robust pipeline for analysis, and cloud operators can identify anomalies that are hard to detect with simple metrics. Early detection of signs of failure can help to evaluate cloud infrastructure performance and provide feedback, enhancing an environment's services to keep up with the promised SLA.

Besides setting a logging pipeline, the next chapter will cover how to conduct benchmarking to gain more insights into the OpenStack infrastructure and evaluating the private cloud capacity.

9

Benchmarking the Infrastructure – Evaluating Resource Capacity and Optimization

"Save for a rainy day."

– Aesop

One of the major challenges when operating a growing infrastructure is keeping different services up and running within the expected defined **Service Level Agreements** (**SLAs**). During the course of production, unexpected issues might arise, even though a proper monitoring solution has been put in place to act proactively when certain situations are detected, such as scaling cluster nodes up and down to accommodate more tenant workloads. In a complex environment such as OpenStack, hitting system performance limits is considered a common issue due to its distributed and loose architecture. The more services join the ecosystem, the more performance issues will be raised. You might have a monitoring dashboard where all services are showing green but cloud tenant users are facing issues with spawning a **Virtual Machine** (**VM**) after several failed requests. Logging, as we covered in the previous chapter, could help with deep diving into the root cause but not with tracing such reluctant events leading to user frustration due to undetected performance issues. For this reason, a best practice is to keep profiling the cloud environment in each minor or major ecosystem. One of the most highly recommended approaches is to include a benchmarking stage in your CI/CD pipeline for each change. Another way is to conduct a profiler cycle on each OpenStack environment software or hardware update and collect performance metrics to compare to the existing SLA. The OpenStack cloud environment does not come with infinite resources, and knowing your limits in advance is essential before making business and operational decisions to extend your service catalogs to cloud users.

In this chapter, we will go through different options for running benchmarking exercises in OpenStack, detecting potential bottlenecks, and working on recommendations to avoid performance issues in production. The following topics will be covered:

- Improve database performance in OpenStack using caching
- Benchmark OpenStack and identify the source of performance bottlenecks
- Optimize our cloud control and data plane using Watcher
- Learn how to trace requests traveling across OpenStack services to identify bottlenecks and improve performance

Empowering the database

Databases in OpenStack are considered one of the most critical shared infrastructure services that require additional attention. From a high-availability perspective, a multi-master cluster based on **Galera MariaDB** would satisfy the requirement for a resilient database setup would does not fully guarantee high-performing write or read transactions. The database performance becomes an issue when the cloud environment keeps growing without measuring the new load. Databases in OpenStack can grow massively and result in large tables. Each read or write request and API call increases the load, leading to common scenarios encountered with databases, such as database inconsistency. A tenant could fire an API request to disassociate a network interface from an instance but the record in the database would remain unchanged. A quick fix is to log in to the database and change the record manually, which can be an error-prone process. The other common pattern is **multiple writes concurrency**, whereby two services assign the same resource ID based on an incoherent status attribute. For example, a new instance might be spawned and may be unable to associate floating IP resources that have been, at the same time, dissociated from a terminated instance. Relational databases typically face performance challenges, and generally, cloud operators utilize database administrators to keep their large and complex database environments. Each request in an OpenStack environment would reach a database for a read or write, and as we have learned in previous chapters, most OpenStack services interact with the database to fulfill a request, ideally within an acceptable response time.

Aside from possible ways to improve an OpenStack database setup through hardware upscaling and upgrades, it is important to keep track of generic metrics of the database, such as the number of reads or writes, input/output storage trends, and error rates. That will help you conclude approximately when the database will present a bottleneck. If the software and configuration tweaks won't help bypass the surge, a hardware upgrade will be more convenient to ensure that there is less trouble in the next production cycle.

Running with caching

One of the fastest ways to ensure an optimally performing database is to look at the hardware running the cluster. The nature of physical disks has a great influence on their capability to perform a certain number of operations. For example, using **Solid State Devices** (**SSDs**) for database nodes can be

very beneficial in improving the access time and transfer data speed, as well as the input/output wait factors. The latest generations will consider Flash storage devices that improve read/write operations and are capable of handling high operation concurrency rates. On the other hand, there are better and more cost-effective ways to design a well-performing database solution, such as by reducing the disk input/output activities on the database side via caching.

Caching happens at every step along the way, from the servers to the browsers of end users. From the end user's perspective, caching can minimize unresponsive statuses when passing queries all the way to the database. Additionally, caching might be suitable for moving a long queue of database queries entirely outside of the database server. In such cases, you are better off looking at an external caching solution, such as **Memcached** or **Redis**. By exposing a memory server, the OpenStack database servers can benefit from a caching layer for Horizon to store OpenStack service data. One important consideration is that the caching layer does not store data. Once, for example, a Memcached instance restarts, the data will be lost. Memcached is a typically adopted caching solution in OpenStack that runs in any type of configuration. Large-scale OpenStack environments run the caching layer in dedicated cluster nodes to double the performance.

A typical Memcached setup only requires the usage of hardware with fewer CPU specifications in contrast to database requirements. The following diagram depicts how Memcached is used in the OpenStack environment:

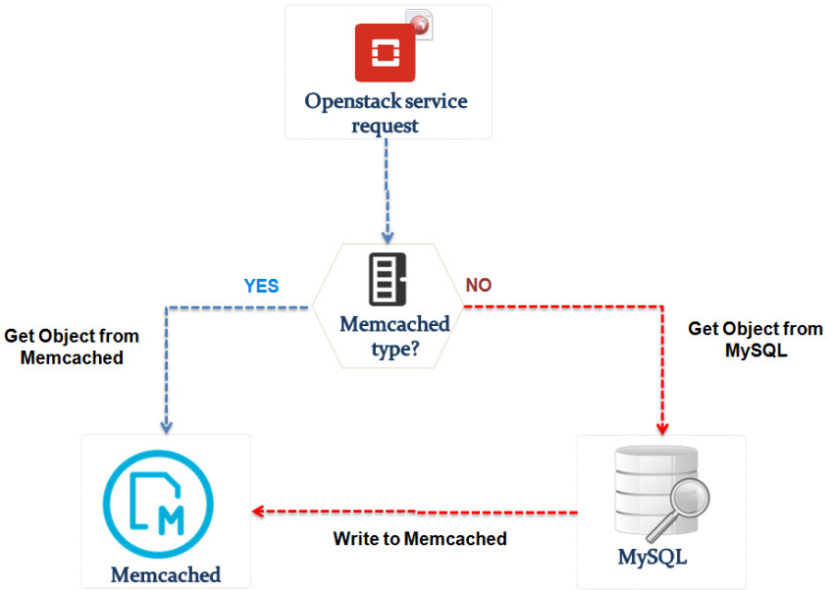


Figure 9.1 – The Memcached integration in OpenStack

This workflow diagram exposes a **write-through caching mechanism** by getting data that is stored in Memcached while it performs a read to the MySQL database. In the next section, we will deploy a Memcached layer in an existing OpenStack environment using `kolla-ansible`.

Deploying caching

Before we start deploying our Memcached instance in the existing OpenStack environment, it is essential to iterate through common workflows in OpenStack that would engage caching operations. When firing an instance creation request, several API requests are generated and aligned to reach specific service endpoints. This process involves requests from Horizon to Nova to launch the API request, Glance to fetch an image, Cinder to attach volumes, Neutron to assign network ports, and so on.

Upon each request, Keystone checks its records in the database to verify the validity of internal tokens.

With a large number of similar API requests, Keystone would consume more CPU resources for the database to fetch those tokens and validate them accordingly. That would increase latency in order to fulfill new incoming requests and lead to longer delays in lookups through the database table caused by expired tokens. Caching can drastically reduce the load on the database by not saving the tokens in the database and instead using the caching layer. Keystone will save all its token records in a Memcached instance.

In the following exercise, we will deploy a Memcached container as part of the control plane nodes. As recommended previously, the Memcached layer can be run on a dedicated server of even clusters for the highest performance and would require clustering configuration separately, which falls outside of the scope of this book. Update the `multi_packtpub_prod` file to assign the `memcached` role to cloud controller nodes, as follows:

```
...
[memcached:children]
control
```

The next step is to enable the Memcached service in the `globals.yml` file:

```
...
enable_memcached: "yes"
```

Finally, run the pipeline and make sure that the `memcached` container is created and up and running by executing the following command line on the controller node:

```
$ sudo docker ps | grep memcached
```

This is the output we get:

```
224fa15acd1e   quay.io/openstack.kolla/memcached:2023.1-ubuntu-jammy      "dumb-init --single-…"   2 hours ago   Up 2 hours (healthy)
```

Figure 9.2 – Listing the Memcached kolla container

Make sure that Keystone has been reconfigured to use caching, which can be checked in the Keystone configuration file in the cloud controller node:

```
$ cat /etc/keystone/keystone.conf
[cache]
backend = oslo_cache.memcache_pool
enabled = True
memcache_servers = url:127.0.0.1:11211
```

Further checks can be performed on the Memcached instance by watching the `get_hits` value increase when firing Keystone API calls, as depicted here:

```
$ watch -d -n 1 'memcached-tool 127.0.0.1:11211 stats'
```

We get the following output:

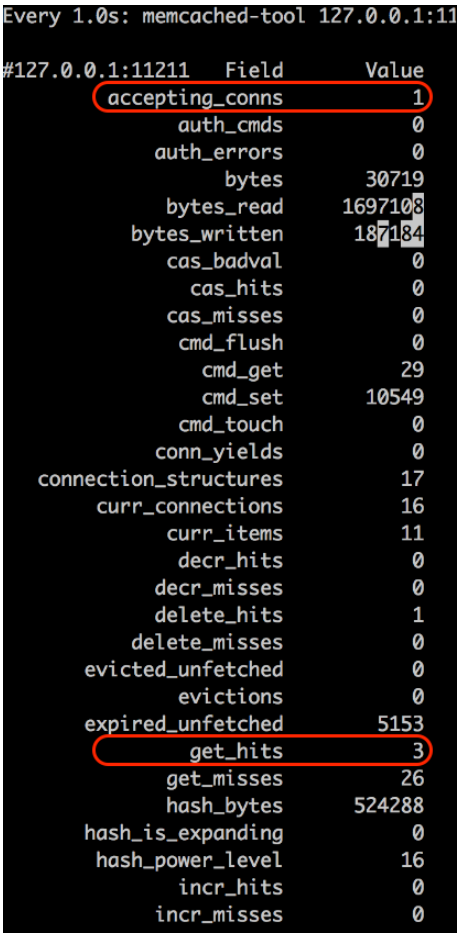


Figure 9.3 – Listing Memcached get_hits stats

A few bounced usage statistic values are useful to verify the current behavior of the Keystone caching mechanism in real time, such as the following:

- `accepting_conns`: This is the number of accepted connections to the Memcached server. Any newly added service configured to use Memcached as a cache backend will increase its value by 1.
- `bytes`: This is the number of bytes used for caching items in real time.
- `bytes_read`: This is the number of incoming bytes to the Memcached server.
- `bytes_written`: This is the number of outgoing bytes from the Memcached server.
- `cmd_get`: This is the number of get commands received by the Memcached server.
- `cmd_set`: This is the number of set commands processed by the Memcached server.
- `get_hits`: This is the number of successful cache hits (get requests). The hit rate can be obtained by dividing `get_hits` by the `cmd_get` value and generating the percentage.
- `get_misses`: This is the number of failed cache hits (get requests).

Most of the OpenStack services can take advantage of the caching layer, so Keystone can use Memcached servers to store the tokens for each service request instead of caching them by default in-process. Once enabled, `kolla-ansible` will roll out the caching layer and adjust the services configuration to use the Memcached layer.

As Memcached is deployed through the cloud controller nodes, we can make sure that its clients are handled by the HAProxy load balancer and use multiple Memcached instances in TCP mode. We can simply edit the `enable_haproxy_memcached` variable in the `kolla-ansible/ansible/group_vars/all.yml` file, as follows:

```
...
enable_haproxy_memcached: "yes"
```

After running the pipeline, the Memcached cluster will be listed in HAProxy and served through its virtual IP.

We will need to tell Nova services that we already have multiple Memcached instances running in three different cloud controller nodes. When `cc01.os` becomes unavailable, `cc02.os` takes over, and so on. We will set the following directive in each controller and compute node in the `/etc/nova/nova.conf` file:

```
...
memcached_servers = cc01.os:11211,cc02.os:11211,cc03.os:11211
...
```

Memcached can also be beneficial for our dashboard. We can tell Horizon to use Memcached for the Django web caching. It just needs to point to the virtual IP, considering a scalable cloud controller setup. The dashboard includes the **CACHES** settings, which we need to edit or add. On your cloud controller nodes, edit the `/etc/openstackdashboard/local_settings.py` file like this:

```
...
 CACHES = {
     'default': {
         'BACKEND':
              'django.core.cache.backends.memcached.'
              'MemcachedCache',
         'LOCATION': '10.0.0.100:11211',
     }
 }
...
```

Optionally, the next configuration snippet can be added to each HAProxy instance to boost a scalable Django dashboard, which will now use a scalable Memcached setup:

```
listen horizon 10.0.0.47:80
balance roundrobin
maxconn 10000
mode tcp
server cc01.os 10.0.0.100:80 cookie cc01.os check inter 5s rise
2 fall 3
server cc02.os 10.0.0.101:80 cookie cc02.os check inter 5s rise
2 fall 3
server cc03.os 10.0.0.102:80 cookie cc03.os check inter 5s rise
2 fall 3
```

Reload the HAProxy configuration by firing the following command line:

```
$ service haproxy reload
```

There are a gazillion use cases where managing the performance of databases is important; it is a large topic and demands more database expertise to find anomalies and tackle them immediately. Memcached is one way to handle massive bursts of read operations, but that may not be enough. Database performance can be more complicated to tackle when it grows and there are few tools available to automate database performance checks and remediation. OpenStack comes with some tooling around its core ecosystem that supports cloud operators to benchmark, observe, and make architectural decisions to improve service performance. In the next section, we will dive into the art of OpenStack benchmarking using automated tooling.

Benchmarking the cloud

Scaling up hardware resources is the standard way to address capacity and performance limits. With the help of a monitoring system, cloud operators can react proactively to add more resources in a defined window of time to accommodate the additional load. However, monitoring systems are not enough to better know our limits. In distributed computing systems, every circulated request incurs a performance hit. In the OpenStack world, a load of API requests can be complicated to trace and develop an approximate measurement of how much a part or service can handle.

From the early stages of the cloud journey, cloud operators should define and develop a strategic approach to measure their cloud limits and performance metrics. However, the challenging part is the absence of efficient tools that could be integrated into the life cycle of cloud deployments.

To address this gap of performance measurement, one key factor is to benchmark the private cloud setup under load at scale for the control and data planes separately. It is heartening to know that with the great success of OpenStack, more benchmarking tools are being developed around its ecosystem and for each plane. In the next section, we will explore a sophisticated benchmarking tool to measure the control plane: the **Rally** tool.

Rally in action

`kolla-ansible` does not support Rally out of the box for installation as Rally is not originally native or part of the OpenStack ecosystem. Installing Rally can be done in various ways, and its package installation is available for many Linux distributions. We will keep our installation platform-agnostic by using containers, but this is not a must.

> **Important note**
>
> You can still integrate the Rally service in the `kolla-ansible` code. You will need to build your own Rally container, push it to your own private repository, and write the Ansible role.

For the next installation exercise, we will use the latest Docker Rally image from Docker Hub that comes with the Rally service and different plugins for OpenStack. In one of the cloud controller nodes, pull the Rally container by running the following command line in a new directory:

```
$ docker pull xrally/rally-openstack
```

The pulled image version of Rally is 2.3.0, which is the latest version at the time of writing. Database, configuration, and records of Rally can be stored by creating a Docker volume as follows:

```
$ docker volume create --name rally_storage
```

The next command step is simply running the Rally Docker container by indicating the path where Rally will persist data under the `/home/rally/.rally` directory:

```
$ docker run -v rally_storage:/home/rally/.rally xrally/rally-
openstack
```

> **Important note**
>
> The Rally default configuration places the `'rally.sqlite'` Rally database under the `/home/rally/.rally` directory. This default configuration can overridden by updating the available options located in the `/etc/rally/rally.conf` file.

We will need to register our OpenStack environment with Rally by providing a deployment file in which OpenStack environment variables, such as admin credentials, are required.

> **Important note**
>
> Different variables of the admin username, password, and tenant are generated in the `/etc/kolla/clouds.yaml` file.

In the Rally container Bash prompt, create a `deployment.json` file and make sure that the different variables are assigned values from the `/etc/kolla/clouds.yaml` file:

```
{
    "type": "cloud_my",
    "auth_url": "http://10.0.0.100:5000/v2.0",
    "region_name": "RegionOne",
    "admin": {
        "username": "admin",
        "password": "476212da112412",
        "tenant_name": "admin"
    }
}
```

Using the Rally client command line, create a deployment using the previously created file:

```
$ rally deployment create --file=deployment.json --name=perf_cloud
```

The output is as follows:

```
2024-03-01 12:31:27.674 93 INFO rally.deployment.engine [-] Deployment eda332da-eeda-31aa-b32a1-ba3126da281 | Starting:  OpenStack cloud deplo
2024-03-01 12:31:27.831 93 INFO rally.deployment.engine [-] Deployment eda332da-eeda-31aa-b32a1-ba3126da281 | Completed: OpenStack cloud deplo
+--------------------------------------+----------------------------+------------+------------------+--------+
| uuid                                 | created_at                 | name       | status           | active |
+--------------------------------------+----------------------------+------------+------------------+--------+
| eda332da-eeda-31aa-b32a1-ba3126da281 | 2024-03-01 12:31:27.579275 | perf_cloud | deploy->finished |        |
+--------------------------------------+----------------------------+------------+------------------+--------+
Using deployment: eda332da-eeda-31aa-b32a1-ba3126da281
~/.rally/openrc was updated
```

Figure 9.4 – Listing the Rally deployment status

Source the generated `openrc` file located under `~/.rally`:

```
$ source ~/.rally/openrc
```

Verify the availability of the OpenStack deployment using the deployment check command, as follows:

```
$ rally deployment check
```

Here is the output:

```
Available services:
+-------------+----------------+-----------+
| Service     | Service Type   | Status    |
+-------------+----------------+-----------+
| __unknown__ | alarming       | Available |
| __unknown__ | placement      | Available |
| __unknown__ | volumev2       | Available |
| __unknown__ | volumev3       | Available |
| ceilometer  | metering       | Available |
| cinder      | volume         | Available |
| cloud       | cloudformation | Available |
| glance      | image          | Available |
| gnocchi     | metric         | Available |
| heat        | orchestration  | Available |
| keystone    | identity       | Available |
| magnum      | container-infra| Available |
| neutron     | network        | Available |
| nova        | compute        | Available |
```

Figure 9.5 – Listing the OpenStack services statuses

Now that we have a Rally server installed and properly configured to talk to OpenStack APIs, it is time for cloud benchmarking. By default, you may find numerous benchmarking scenarios under `/rally/sample/tasks/scenarios` for all OpenStack services, including other incubated projects such as Murano and Sahara. We will concentrate on benchmarking our existing running

OpenStack services. Before starting our first benchmark test, it would be great to shine the spotlight on how Rally works in the first place. Scenarios in Rally are performed based on tasks. A task can include a set of running benchmarks against the OpenStack cloud written in sample JSON or YAML file format. The former file generally has the following structure:

```
ScenarioClass.scenario_method:
-
args:
...
runner:
...
context
...
sla:
...
```

Each stanza block is defined as follows:

- `ScenarioClass.scenario_method`: This defines the name of the benchmark scenario.
- `args`: Every method corresponding to a specific class scenario can be customized by passing parameters before launching the benchmark.
- `runner`: This defines the workload frequency type and the order of the benchmarking scenarios. The runner stanza can support different types, as follows:
  - `constant`: This involves running the scenario for a fixed number of times. For example, a scenario can be run 10 times in the total test period.
  - `constant_for_duration`: This involves running the scenario for a fixed number of times until a certain point in time.
  - `periodic`: This involves defining a certain period to run two consecutive benchmark scenarios.
  - `serial`: This involves running the scenario for a fixed number of times in a single benchmark thread.
- `context`: This defines the environment type in which our benchmark scenario(s) can run. Usually, the concept of context defines how many tenants and active users will be associated with a given OpenStack project. It can also specify a quota per tenant or user within a certain granted role.
- `sla`: This is very useful for identifying the overall scenario average success rate of the benchmark.

If you are hoping to find a convenient benchmarking scenario that will reveal more significant results from your current OpenStack deployment, you'll have to keep looking for a real use case that is more specific to cloud operators. For example, Rally can help developers easily run synthetic workloads such as VM provisioning and destroy instances for a limited period. However, the case seems to be more complicated for cloud operators. Such results generated from workloads are more high-level but allow you to identify bottlenecks in the cloud.

Let's consider a real-world example: companies have several applications that need to be deployed in different usage patterns. If we have multiple concurrent instances of an application for QA/dev, they will be deployed in different versions of this application on the cloud several times per day. Let's take the use case of a large deployment, where there is a set number of teams running a bunch of standard stack applications and each application will contain a lot of VMs that need to be deployed at certain times in a day. Such workload requirements are translated to OpenStack terms as follows: we will have *M* number of users provisioning *N* number of VMs within a specific flavor and time period in a concurrent way. As we know, OpenStack is not a monolithic structure; it is a distributed system with different daemons and services talking to each other. If we decompose a use case of provisioning an instance to the primitives, it helps us to understand where we spend most of the time during the VM provisioning phase and build records of historical data. For example, by running the same benchmark several times but changing the parameters of database configuration or enabling Glance. In the next subsection, we will use the same benchmarking approach to run against the Keystone service.

Keystone under stress

Our example scenario is a benchmarking test based on the Rally method named `KeystoneBasic.authenticate_user_and_validate_token`. The scenario is intended to measure the time to fetch and validate tokens issued by Keystone when authenticating users under a specific load. An important note for the sake of the demonstration is that the load test will be applied on a specific Keystone configuration that does not support the **WSGI** module for the Apache web server.

> **Important note**
>
> The latest version of `kolla-ansible` comes with Keystone configured with the WSGI (`mod_wsgi`) module enabled by default for Apache. You can disable the module manually in the Keystone configuration file or create a new Kolla container to test the first Rally scenario. Make sure to perform the load exercise in a separate environment to not break the Keystone configuration and hence other services.

Let's create a new file named `perf_keystone_pp.yaml`. The content of the file task looks as follows:

```
KeystoneBasic.authenticate_user_and_validate_token:
- args: {}
  runner:
    type: "constant"
    times: 50
```

```
    concurrency: 50
  context:
    users:
      tenants: 5
      users_per_tenant: 10
  sla:
failure_rate:
  max:1
```

The sample scenario will create a constant load of a Keystone scenario authenticating users and validating tokens 50 times without pausing by creating 5 different tenants with 10 users in each. Note that in each iteration, 50 scenarios will run at the same time in concurrency mode to simulate multiple user access. The `sla` section defines a condition where if one authentication attempt fails, then the task will be aborted.

Let's run the previous benchmark using the Rally command line, as follows:

```
$ rally task start --abort-on-sla-failure perf_keystone_pp.yaml
```

Note that this time, we added a new option to our `abort-on-sla-failure` command. This is a very useful argument if you are running such a benchmark scenario in a real OpenStack production environment. Rally generates a heavy workload, which might cause performance troubles in the existing cloud. Thus, we tell Rally to stop the load at a certain moment when the `sla` conditions are met. The output of our executed task is as follows:

```
+-------------------------------------------------------------------------------------------------------------------+
|                                            Response Times (sec)                                                   |
+---------------------------+-----------+--------------+--------------+--------------+-----------+---------+-------+
| Action                    | Min (sec) | Median (sec) | 90%ile (sec) | 95%ile (sec) | Max (sec) | Avg (sec) | Success | Count |
+---------------------------+-----------+--------------+--------------+--------------+-----------+-----------+---------+-------+
| keystone_v2.fetch_token   | 5.379     | 25.036       | 41.745       | 48.185       | 57.831    | 24.286    | 100.0%  | 50    |
| keystone_v2.validate_token | 11.575    | 34.219       | 53.503       | 55.351       | 56.679    | 37.924    | 100.0%  | 50    |
| total                     | 35.014    | 57.384       | 88.948       | 92.12        | 100.932   | 62.211    | 100.0%  | 50    |
+---------------------------+-----------+--------------+--------------+--------------+-----------+-----------+---------+-------+

Load duration: 195.66706
Full duration: 405.882441
```

Figure 9.6 – Listing Rally task stats

The Rally benchmark results show that the scenario test has run 50 times and is completed with a 100% success rate. To dive into more details, we can visualize the HTML report using the generated Rally task ID by running the following command:

```
$ rally task report c5493ee7-fba2-4290-b98c-36e47ed0fdb2 --out
/var/www/html/bench/keystone_report01.html
```

Our first test iteration benchmark involves a simple SLA condition that was met during the Rally task:

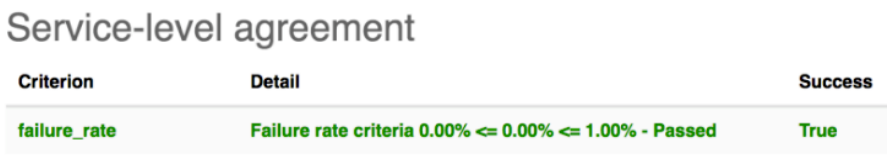


Figure 9.7 – A Rally SLA failure_rate HTML report

From the same report dashboard, in the **Overview** tab, a second pertinent chart, **Load Profile**, illustrates how many iterations were running in parallel during the Rally task:

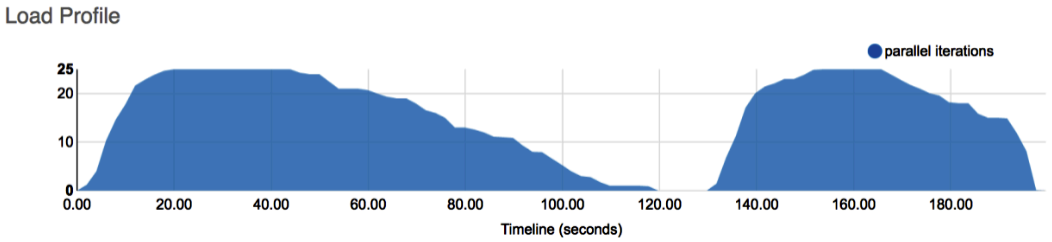


Figure 9.8 – The Load Profile report for the Keystone scenario, generated by Rally

The **Load Profile** graph can be used to illustrate the variation of running iterations simultaneously over the workload period. This information is useful for learning about the system behavior at certain peaks and planning how much load can be supported at any given time. More details are provided in the second tab, **Details**, where we can find **Atomic Action Durations** charts showing, in our case, two actions – **keystone_v2.fetch_token** and **keystone_v2.validate_token**:

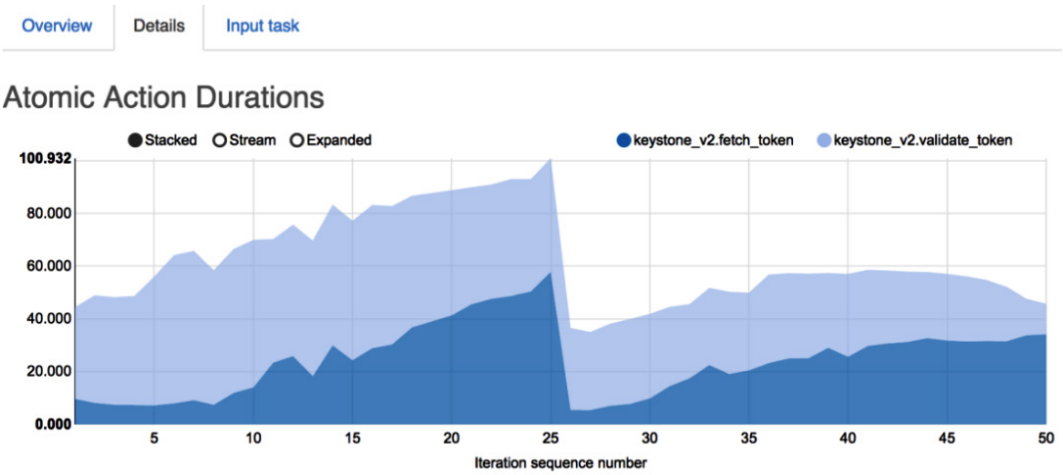


Figure 9.9 – An Atomic Action Durations graph for Keystone steps, generated by Rally

The chart helps with seeing the variation of the scenario for each action and how the duration is affected and changed throughout the execution of iterations. As we can see, both actions do not have the same duration as fetching and validating tokens are two different operations. If our test case failed in terms of SLA conditions, such as a very long duration for scenario execution, we can use this chart to drive a granular analysis of which action the bottleneck occurred on.

We can adjust our success criteria parameters a bit in a second iteration for a stricter SLA to visualize a more realistic scenario. For example, we can modify our `sla` section as follows:

```
...
  sla:
    max_avg_duration: 5
    max_seconds_per_iteration: 5
  failure_rate:
  max: 0
  performance_degradation:
    max_degradation: 50
    outliers:
      max: 1
```

The new `sla` section defines five conditions:

- `max_avg_duration`: If the maximum average duration of an authentication is longer than five seconds, the task will be aborted.
- `max_seconds_per_iteration`: If the maximum duration of an authentication request is longer than five seconds, the task will be aborted
- `failure_rate`: If more than one authentication fails, the task will be aborted defined by the parameter `max`.
- `performance_degradation`: If the difference between the maximum and minimum duration of completed iterations is more than 50 percent, the task will be aborted. The maximum value is defined by `max_degradation` parameter.
- :`outlier`: The outlier limits the number of long-running iterations to a value of `1` defined by the `max` parameter.

Once the Rally scenario has been modified, rerun the task as follows:

```
$ rally task start --abort-on-sla-failure perf_keystone_pp.yaml
```

Let's check our charts again by generating a new report with a different name so we can compare the difference in the results from the previous iteration:

```
$ rally task report 980957ef-4c4c-4e9b-a9c1-573839dcad80 --out
/var/www/html/bench/keystone_report02.html
```

Service-level agreement

Criterion	Detail	Success
performance_degradation	Current degradation: 4937.442268% - Passed	True
max_seconds_per_iteration	Maximum seconds per iteration 11.27s ◀— 5.00s - Failed	False
failure_rate	Failure rate criteria 0.00% ◀— 0.00% ◀— 1.00% - Passed	True
outliers	Maximum number of outliers 0 ◀—1 - Passed	True
aborted_on_sla	Task was aborted due to SLA failure(s)	False

Figure 9.10 – An SLA for the Keystone load profile, generated by Rally

During the test, Rally detected a maximum value for the iteration of `11.27` seconds, which does not comply with our SLA requirement:

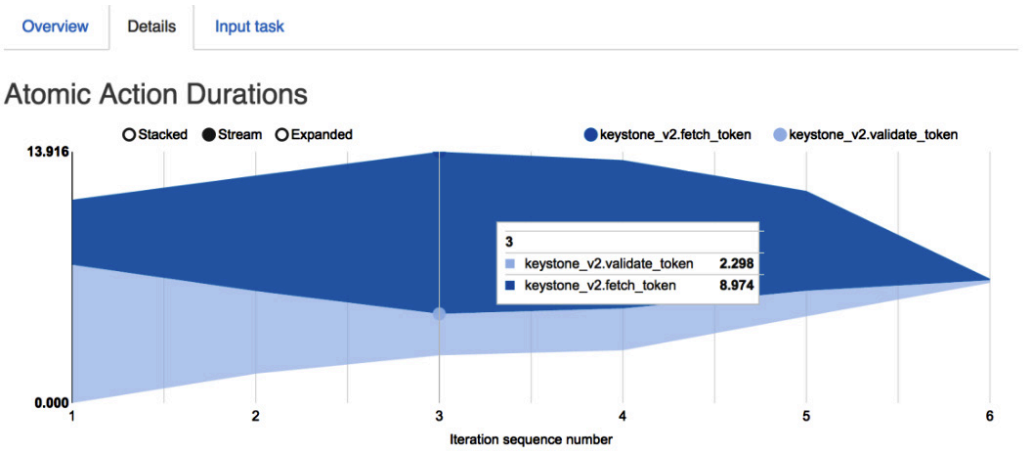


Figure 9.11 – A stacked overview of Keystone actions' durations

With the new SLA conditions, Rally execution stopped at the sixth iteration. The demanded average time of authenticating a user and validating the token was not met and hence, this will affect the overall scenario execution duration time. The next goal is to compete against that value and decrease it to below five seconds. Our benchmark test showed that authenticating and validating user tokens at a certain peak of workload would not achieve our SLA requirements. Moreover, the time spent authenticating one user increases and might be timed out as concurrency levels hit a specific threshold. This performance challenge can be tweaked by revisiting our Keystone setup. We can refer to an advanced Keystone design pattern that empowers our identity service performance within the OpenStack environment.

As many OpenStack components are developed to support the `eventlet-based` process, the Keystone component can run in different ways by supporting the multi-threading process at the cost of our cloud controller's CPU power. One recommendation is to deploy Keystone in an Nginx server under WSGI or an Apache server with the `mod_wsgi` module enabled.

Fronting our Keystone instance with a web server will bring facilities for handling parallel HTTP connections and advanced features to proxy authentication requests to our identity instance in a multi-threaded-based process mode. By default, the community `kolla-ansible` code comes with the WSGI module enabled that will be used for this exercise. The WSGI Keystone configuration is provided in the `wsgi-keystone.conf.j2` template located under the`/ansible/roles/keystone/templates` directory.

The following snippet shows a basic WSGI configuration in Keystone with the `VirtualHost` option:

```
...
<VirtualHost *:{{ keystone_public_listen_port }}>
    ServerName {{ keystone_public_url }}
    WSGIDaemonProcess keystone-public processes={{ keystone_
api_workers }} threads=1  user=keystone group=keystone display-
name=keystone-public
    WSGIProcessGroup keystone-public
    WSGIScriptAlias / {{ binary_path }}/keystone-wsgi-public
    WSGIApplicationGroup %{GLOBAL}
    WSGIPassAuthorization On
...
```

Note that the WSGI process group defines the number of threads to be run by the Keystone user (`threads=1`) and processes (`keystone_api_workers`). We can increase the number of processed requests by adjusting the number for processes defined in the `keystone_api_workers` option and threads in `WSGIDaemonProcessdirective`:

```
...
WSGIDaemonProcess keystone-public processes={{ keystone_api_workers }}
threads=30  user=keystone group=keystone display-name=keystone-public
...
```

As we have introduced a configuration change, run the pipeline to pull and run the new configurations to be reflected in the web server.

Now we have Keystone backed by a web server and empowered by multi-threaded process mode by means of the WSGI module. We have already defined the process daemons and threads that will help trace the limit of our hardware and Keystone capabilities against the same scenario by running it once again:

```
# rally task start --abort-on-sla-failure perf_keystone_pp.yaml
```

Demonstrating the last Keystone settings, our cloud controller should be running more `apache2` processes and hence using more CPU power. That can be illustrated in the new performance results for authentication and validation duration change:

```
# rally task report b36e4b72-0e86-4769-965d-cbe3662d647e --out
/var/www/html/bench/keystone_report03.html
```

Service-level agreement

Criterion	Detail	Success
performance_degradation	Current degradation: 806.294481% - Passed	True
max_seconds_per_iteration	Maximum seconds per iteration 4.19s <= 5.00s - Passed	True
failure_rate	Failure rate criteria 0.00% <= 0.00% <= 1.00% - Passed	True
outliers	Maximum number of outliers 1 <= 1 - Passed	True

Figure 9.12 – The Keystone SLA upon web server threads increasing

Now we have reached our goal by reducing the max number of seconds per iteration to below five seconds (4.19 seconds). As we have achieved a *green* SLA, we can analyze our new Keystone boost configuration from the **Load Profile** chart:

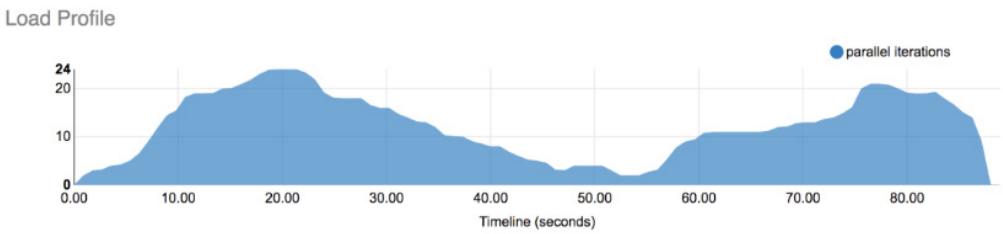


Figure 9.13 – Keystone load profile after increasing number of web server threads

Compared to previous iterations, simultaneous requests can be handled by a pool of threads, so Keystone is able to face concurrent iterations during the workload timeline. Although the currency level was set higher, we can notice that the maximum value of the load testing performed was only **24**. That confirms that our thread and process settings for WSGI were heading in the right direction, leaving more free slots for more concurrency and less time for processing per iteration.

There is more that can be done with Rally, but the scenarios described are sophisticated enough to start our benchmarking journey in OpenStack. Rally is also a pluggable platform that allows operators to create and customize their benchmarking scenarios to meet certain special use cases. Benchmarking is helpful to get insights into how your OpenStack is performing in regard to the defined SLA. However, when performance anomalies are detected, benchmarking cannot help directly to remediate the issue. For this reason, the profiling practice covered in the next section should be taken into consideration.

Profiling the cloud

The OpenStack ecosystem is composed of multiple services that connect with each other to fulfill a request. In some cases, a request can be processed extremely slowly and fail. If the rate of request failures keeps increasing, cloud operators must investigate and understand the root cause of the issue. Monitoring, debugging, and logging tools can partially be the answer to such scenarios, but they lack a request flow mechanism. A tiny but great tool has been developed in the OpenStack ecosystem named **OSProfiler**, which focuses on service tracing. OSProfiler provides a view of requests as they travel through different OpenStack services and compiles data to be visualized in a timeline graph. Cloud operators can determine bottlenecks in the OpenStack deployment and improve its performance by comparing trace sets with different conditions. The OSProfiler tool is capable of providing valuable insights and tracing data that captures the response time of APIs, databases, drivers, and RPC calls. Cloud operators can store the tracing information in persistent storage for further analysis, such as Redis, Elasticsearch, a simple file, or MongoDB.

In a complex system such as OpenStack, tracing where a request is spent most of the time is extremely helpful to troubleshoot faster, identify bottlenecks, and prevent the issue from happening again. Using OSProfiler, cloud operators can figure out the reason, for example, why launching a VM request takes ages. OSProfiler is capable of showing the services involved in such requests, as well as their dependencies. Operators can conclude which service presents the bottleneck and how to improve the response time.

The next section will illustrate how to install and run OSProfiler in the existing OpenStack environment.

Profiler in action

Since the Antelope version, OSProfiler can trace all the OpenStack core services in addition to other projects. The OpenStack community aims to use OSProfiler for all OpenStack projects without exception due to its light and powerful tracing capabilities. The `kolla-ansible` infrastructure already supports OSProfiler, which can be installed easily. OSProfiler can be enabled by configuring the following settings in the `globals.yml` file:

```
enable_osprofiler: "yes"
enable_elasticsearch: "yes"
```

Note that OSProfiler uses Elasticsearch as a backend to store the tracing data.

> **Important note**
>
> If `enable_elasticsearch` is set to `"no"`, Kolla will install and use Redis as a default backend for OSProfiler.

OSProfiler does not need to be installed on a dedicated host. At the time of writing, OpenStack `kolla-ansible` supports OSProfiler for Keystone, Nova, Glance, Cinder, Neutron, Placement, Swift, Heat, Trove, Senlin, and Vitrage.

Once the `globals.yml` file is updated and pushed, run the pipeline and the OSProfiler library should be installed. Before starting a profiling exercise, we will need to retrieve the generated OSProfiler secret referenced by the `osprofiler_secret` key in the `/etc/kolla/passwords.yml` file. The `osprofiler_secret` key will be used to create profiler UUIDs with OpenStack client command lines for supported services. The next example illustrates a tracing exercise of a Glance API call, as demonstrated in the following steps:

1. Use the official OpenStack Python client running in a cloud controller node to generate a tracing output:

```
$ openstack --os-profile <OSPROFILER_SECRET> image list
```

This is the output we get:

```
+--------------------------------------+--------------------------------+--------+
| ID                                   | Name                           | Status |
+--------------------------------------+--------------------------------+--------+
| 5234ed12-ed11-4311-9822-762da54e7def | myPPCirros                     | active |
+--------------------------------------+--------------------------------+--------+
```

Figure 9.14 – OSProfiler image list

`<OSPROFILER_SECRET>` is the retrieved OSProfiler secret retrieved from the `/etc/kolla/passwords.yml` file referenced in the `osprofiler_secret` parameter.

> **Important note**
>
> It is possible to customize the `OSPROFILER_SECRET` value for each supported service by configuring the `hmac_keys` setting in each configuration file that corresponds to that service.

The command line returns an OSProfiler tracing command line, as follows:

```
Display trace with command:

osprofiler trace show --html 75da221e-ffe1-7dac-aae7-22eaed337761
```

Figure 9.15 – An OSProfiler trace command output

2. Use the `osprofiler` command line to print the tracing graph in HTML format. The retrieved tracing data can be stored locally in the created Elasticsearch instance:

```
$ osprofiler trace show --html 75da221e-ffe1-7dac-aae7-
22eaed337761 --connection-string elasticsearch://localhost:9200
--out "/tmp/image_perf.html"
```

An HTML report should be available locally under the `/tmp` directory.

3. Visualize the tracing results by accessing the generated `image_perf.html` file in a browser:

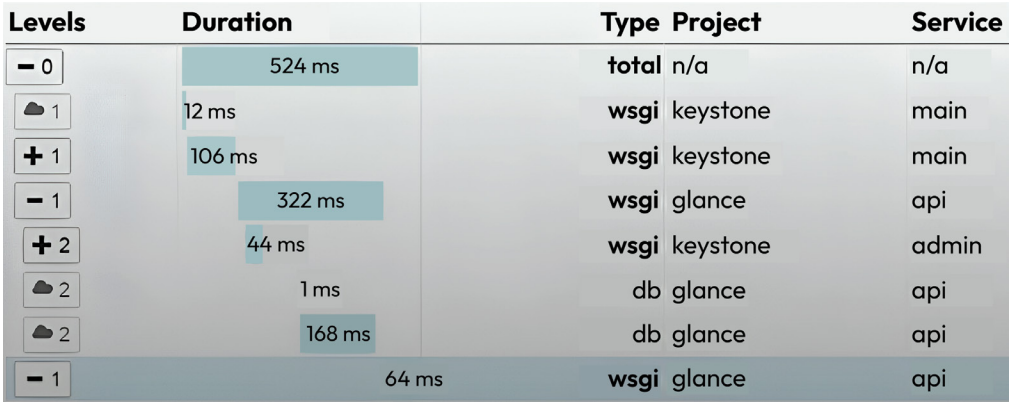


Figure 9.16 – An example of OSProfiler tracing results

The generated HTML report exposes, for each request made at the time of processing of each call, the nature of the service call (API, database, etc.) and its corresponding project. The `Levels` column corresponds to a traced point that includes a low level of detailed information in JSON format.

Tracing is a great practice to debug at a low level what cannot be detected in your monitoring system or overlooked in the log data. Both profiling and benchmarking provide sufficient information about cloud performance and the trend of resource utilization. As the cloud infrastructure keeps receiving new tenants, more hardware demands will increase. If this is not watched and analyzed from the early days, the cost can rise and the risks will be out of control. Operators should be provided with strategies and tooling to automate budget control and management. One of the best approaches with limited resources is to seek optimization, which will be covered in the next section.

Watching the cloud

Once your cloud environment has started to expand and more hardware resources are deployed, cloud operators should find ways to optimize costs. There are niche use cases, such as optimizing the placement of VMs (VMs migrate between hosts in case of imbalance detection).

The lack of tooling and efficient processes to conduct a resource optimization exercise presents a big challenge to private cloud operators. Traditionally, OpenStack administrators needed to manually grab historical metrics on resource usage, analyze them in regular sprints, and make decisions based on the collected data. However, manual processing in a large OpenStack deployment can be error prone. To automate such a process, the OpenStack community has come up with an evolving incubated project code named **Watcher**. The main objective of the Watcher project is to enable new ways for cloud operators to reduce the cloud's **Total Cost of Ownership** (**TCO**). Watcher is designed to monitor, analyze, and execute optimization tasks based on predefined goals. The following diagram illustrates how Watcher's continuous optimization loop is designed through a set of steps:

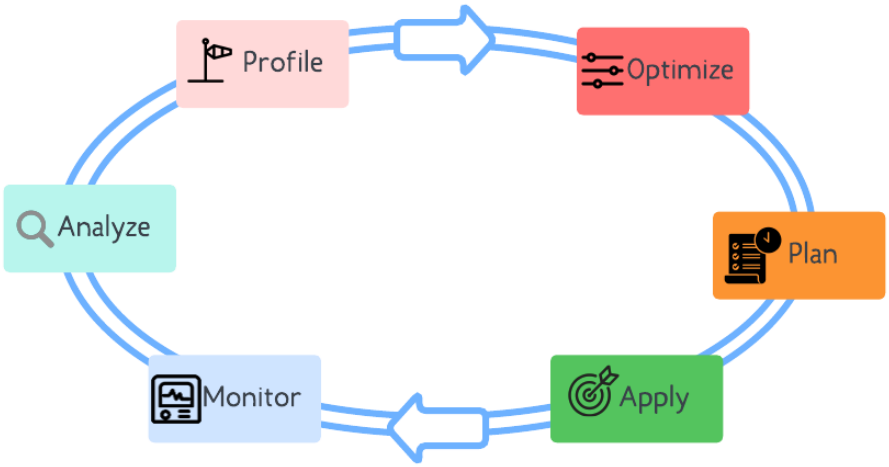


Figure 9.17 – OpenStack's Watcher steps workflow

It starts with the **Monitor** state, where data metrics are collected from various data sources, such as the type of CPU, memory utilization, and energy consumption. Then, the collected information is analyzed and aggregated. Watcher then engages a profiler component that concludes a few patterns and uses that to predict VM resource utilization. An optimization plan will be created by an **Optimizer** component by taking a set of goals and constraints as inputs, for example, by creating some collocation rules such as affinity and anti-affinity rules in the Nova scheduler. Watcher will make scheduling decisions based on such inputs and can leverage some of the other constraints defined by other projects. The next stage runs the **Planner** component, so Watcher builds action items, for example, if a VM needs to be migrated from one host to another. In the Plan phase, Watcher puts a set of steps that can be acted upon and ordered to run serially or in parallel. Finally, Watcher executes the action plan and applies the optimal state of the infrastructure as specified in the defined goals.

The next section will go through the installation of the Watcher project in the existing OpenStack environment using `kolla-ansible` and demonstrate its usage.

Watcher in action

The latest `kolla-ansible` code releases come with the Watcher playbook ready for installation. Keystone, Nova, and Ceilometer must be running before installing Watcher. In the following setup, all Watcher components, including `watcher-api`, `watcher-engine`, and `watcher-applier`, will be part of the cloud controller node group. To install `watcher` components, configure the `multi_packtpub_prod` inventory file by adding the following section:

```
...
[watcher:children]
control
[watcher-api:children]
```

```
watcher
[watcher-engine:children]
watcher
[watcher-applier:children]
watcher
```

Next, enable the Watcher service in the `globals.yml` file by adjusting the following configuration line:

```
enable_watcher: "yes"
```

The Watcher service comes with a Horizon plugin that will be visible on the Horizon dashboard once installed. Commit the changes and run the pipeline. Once finished, check the new Docker containers for the Watcher service by running the following command line in any of the controller nodes:

```
$ sudo docker ps | grep watcher
```

Here is the output:

```
d4aea2172133   quay.io/openstack.kolla/watcher-api:2023.1-ubuntu-jammy           "dumb-init --single-…"   1 hours ago   Up 29
97265dfe31a1   quay.io/openstack.kolla/watcher-engine:2023.1-ubuntu-jammy        "dumb-init --single-…"   1 hours ago   Up 30
99875dcaef52   quay.io/openstack.kolla/watcher-applier:2023.1-ubuntu-jammy       "dumb-init --single-…"   1 hours ago   Up 30
652daca31a76   quay.io/openstack.kolla/watcher-base:2023.1-ubuntu-jammy          "dumb-init --single-…"   1 hours ago   Up 30
```

Figure 9.18 – A Watcher deployment check

Before starting to use the Watcher service, it is important to highlight the different steps of a Watcher workflow:

1. Create an optimization goal and associate it with a strategy.
2. Create an audit template that is associated with the optimization goal.
3. Create an audit that will be triggered by the audit template.
4. Generate an action plan by the created audit.
5. Take action either manually or in an automatic fashion.

In the following part of this section, we will demonstrate a predefined Watcher strategy, which is referred to as the **VM Workload Consolidation Strategy**. The goal of such a strategy is to consolidate a running workload in the cloud and optimize the number of resources that run it.

To quickly check the predefined list of goals in Watcher, run the following command line and take note of the **Server Consolidation** goal **UUID** or **Name** in the output:

```
$ openstack optimize goal list
```

The output is as follows:

```
+--------------------------------------+----------------------+----------------------+
| UUID                                 | Name                 | Display name         |
+--------------------------------------+----------------------+----------------------+
| 553412a2-cba1-652e-cd3a-872da2141ca1 | dummy                | Dummy goal           |
| 8726da12-01da-42b5-65da-86165daca514 | server_consolidation | Server Consolidation |
| 65daa212-8790-8182-ad33-98dcaea133ea | workload_balancing   | Workload Balancing   |
| fada125f-dca1-87da-cda1-9875612da212 | thermal_optimization | Thermal Optimization |
| 76dad123-ffe1-4f0f-dca1-b828464b3fd0 | airflow_optimization | Airflow Optimization |
| 23da1dad-cd4a-88da-98da-cda2231209da | unclassified         | Unclassified         |
+--------------------------------------+----------------------+----------------------+
```

Figure 9.19 – Watcher optimization goal listing

Next, obtain the list of available strategies for the **Server Consolidation** goal using the listed goal **UUID** or **Name** from the previous output:

```
$ openstack optimize strategy list --goal 8726da12-01da-42b5-65da-
86165daca514
```

We get the following output now:

```
+--------------------------------------+---------------------------+-----------------------------------+----------------------+
| UUID                                 | Name                      | Display name                      | Goal                 |
+--------------------------------------+---------------------------+-----------------------------------+----------------------+
| eada1312-3314-4da1-ddac-7235da3e4176 | basic                     | Basic offline consolidation       | server_consolidation |
| cdc1244d-0eb2-3312-9871-987daca123da | vm_workload_consolidation | VM Workload Consolidation Strategy | server_consolidation |
| 413ea121-1123-7611-dca1-01031217761a | node_resource_consolidation | Node Resource Consolidation strategy | server_consolidation |
+--------------------------------------+---------------------------+-----------------------------------+----------------------+
```

Figure 9.20 – A Watcher optimization strategy listing

From the returned list of strategies, we will use the `vm_workload_consolidation` strategy to minimize the number of hosts running a workload with respect to resource capacity constraints. Before creating the audit template, we can check the distribution of the instances across different hypervisor machines before applying the selected optimization strategy in the Watcher panel:

Hypervisor | Compute Host

Displaying 3 items

Hostname	Type	VCPUs (used)	VCPUs (total)	RAM (used)	RAM (total)	Local Storage (used)	Local Storage (total)	Instances
cn01.os	QEMU	12	16	12GB	25.3GB	125.3GB	544GB	2
cn02.os	QEMU	18	16	25GB	25.3GB	436GB	544GB	18
cn03.os	QEMU	11	16	16GB	25.3GB	105GB	544GB	10

Figure 9.21 – The instance distribution state before the optimization is applied

The **Hypervisor** list is composed of three compute nodes and the specific distribution of instances spread across them.

In the next step, create an audit template associated with the selected goal and the chosen strategy by running the following command line:

```
$ openstack optimize audittemplate create consolidation_template
8726da12-01da-42b5-65da-86165daca514 --strategy cdc1244d-0eb2-3312-
9871-987daca123da
```

Here is the output:

```
+-------------+--------------------------------------+
| Field       | Value                                |
+-------------+--------------------------------------+
| UUID        | 123761da-ed10-dd11-43da-ca34eeda1251 |
| Created At  | 2024-02-26T18:21:19.541278+00:00     |
| Updated At  | None                                 |
| Deleted At  | None                                 |
| Description | None                                 |
| Name        | consolidation_template               |
| Goal        | server_consolidation                 |
| Strategy    | vm_workload_consolidation            |
| Audit Scope | []                                   |
+-------------+--------------------------------------+
```

Figure 9.22 – Watcher optimization audit template creation

Next, run an audit from the created audit template, as follows:

```
$ openstack optimize audit create -a consolidation-template
```

The output is as follows:

```
+---------------+-----------------------------------------------------+
| Field         | Value                                               |
+---------------+-----------------------------------------------------+
| UUID          | da23a982-deaa-3231-3120-237dea123cba                |
| Name          | vm_workload_consolidation-2024-02-26T18:23:19.5412  |
| Created At    | 2024-02-26T18:21:19.541278+00:00                    |
| Updated At    | None                                                |
| Deleted At    | None                                                |
| State         | PENDING                                             |
| Audit Type    | ONESHOT                                             |
| Parameters    | {'period': 3600, 'granularity': 300}                |
| Interval      | None                                                |
| Goal          | server_consolidation                                |
| Strategy      | vm_workload_consolidation                           |
| Audit Scope   | []                                                  |
| Auto Trigger  | False                                               |
| Next Run Time | None                                                |
| Hostname      | None                                                |
| Start Time    | None                                                |
| End Time      | None                                                |
| Force         | False                                               |
+---------------+-----------------------------------------------------+
```

Figure 9.23 – Watcher optimization audit creation

Note that the audit creation may take longer to complete. The **State** field shows **PENDING**. During that state, the audit request is treated by the Watcher decision engine, which passes the requested audit to the next state. If the Watcher decision engine finds at least one optimization option that can be applied within the set of the audited resources (compute nodes for the context of the current goal and strategy), the audit state will change to **SUCCEEDED**.

Use the `optimize audit show` command with the generated audit UUID and verify whether the status of the audit creation is completed:

```
$ openstack optimize audit show da23a982-deaa-3231-3120-237dea123cba
```

Here is the output:

```
+---------------+-----------------------------------------------------+
| Field         | Value                                               |
+---------------+-----------------------------------------------------+
| UUID          | da23a982-deaa-3231-3120-237dea123cba                |
| Name          | vm_workload_consolidation-2024-02-26T18:23:19.5412  |
| Created At    | 2024-02-26T18:23:37+00:00                           |
| Updated At    | 2024-02-26T18:45:16+00:00                           |
| Deleted At    | None                                                |
| State         | SUCCEEDED                                           |
| Audit Type    | ONESHOT                                             |
| Parameters    | {'period': 3600, 'granularity': 300}                |
| Interval      | None                                                |
| Goal          | server_consolidation                                |
| Strategy      | vm_workload_consolidation                           |
| Audit Scope   | []                                                  |
| Auto Trigger  | False                                               |
| Next Run Time | None                                                |
```

Figure 9.24 – Watcher optimization audit validation

For each created audit, the optimization service generates an action plan. An action plan defines which tasks need to be executed to achieve the goal set at the start. Action plans use advanced algorithms and return some pertinent information, such as **efficacy indicators** (reflects an improvement score based on the generated audit solution) and global efficacy (general scoring for the evaluated action plan).

Once the audit creation status changes to **SUCCEEDED**, run the following command line with the audit UUID to retrieve the action plan generated by the audit:

```
$ openstack optimize actionplan list --audit da23a982-deaa-3231-3120-
237dea123cba
```

The output is as follows:

```
+--------------------------------------+--------------------------------------+-----------+------------+-------------------------------+
| UUID                                 | Audit                                | State     | Updated At | Global efficacy               |
+--------------------------------------+--------------------------------------+-----------+------------+-------------------------------+
| 23a766da-77da-2361-bca1-efeda124870d | da23a982-deaa-3231-3120-237dea123cba | RECOMMENDED | None       | Released_nodes_ratio: 33.00 % |
|                                      |                                      |           |            |                               |
+--------------------------------------+--------------------------------------+-----------+------------+-------------------------------+
```

Figure 9.25 – Watcher optimization action plan listing

Note in the previous output that the Watcher planner has created an action plan that is waiting for operator validation. The **RECOMMENDED** state is the last step performed by the Planner component. From the **Global efficacy** information, the `watcher-planner` process has concluded that around 33% of compute nodes can be pulled back while maintaining the same amount of running workloads.

The returned action plan line is not sufficient to go through all the different action items. There is a separate command line for that purpose. Before applying any action plan, an operator should review the associated actions as follows. Run the following command line with the action plan UUID generated in the previous step:

```
$ openstack optimize action list --action-plan 23a766da-77da-2361-
bca1-efeda124870d
```

The action plan lists all items that will be actioned. The following list is a truncated version for a longer output that indicates the change of the Nova service's state in the first line. The following lines are the planned action based on the first item that will be executed serially. Each following action item is a recommendation of the Watcher planner to migrate a set of instances once the parent action (the Nova service's state) is updated:

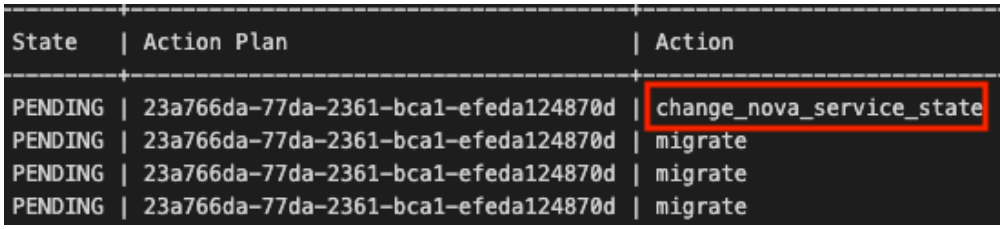


Figure 9.26 – Watcher optimization action listing

Once reviewed, an operator can go ahead and run the `watcher-applier` process to execute the action plan by running the following command line with the action plan UUID:

```
$ openstack optimize actionplan start 23a766da-77da-2361-bca1-
efeda124870d
```

The output can be quite large due to the values assigned to action plan indicators (**Efficacy indicators** and **Global efficacy** embedded JSON values). A simple truncated version can be visualized as follows:

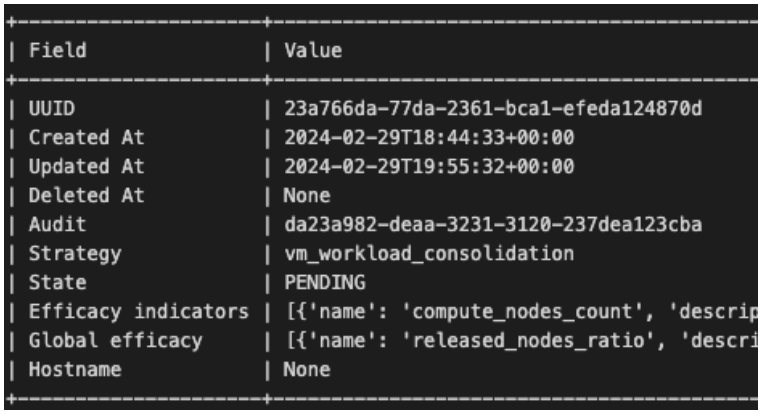


Figure 9.27 – Watcher optimization action start

Most importantly, note the **PENDING** state of the action plan execution. That is expected to keep running for a longer time due to the migration of a set of instance tasks in the background. The **Efficacy indicators** and **Global efficacy** fields simply reflect a description of the recommended action plan that can be summarized as follows:

- **Efficacy indicators**: This refers to the number of compute nodes in which the **Applier** component is performing optimization actions. In our example, we are dealing with three nodes. As recommended, the audit aims to release one compute node from the three. Additionally, the indicator keeps counting each migrated instance reported by the Nova service when accomplished. The `instance_migrations_count` recommendation indicates that the number of instance migrations to be performed is `2`.
- **Global efficacy**: This is the ratio of nodes aimed to be released once all action items are executed. The ratio is obtained by dividing the number of released compute nodes by the total number of compute nodes in the audit scope. In our case, the ratio will be approximately 33%.

You can run the previous command line several times and notice the change of the action plan execution state from **PENDING** to **ONGOING**. Once all migration tasks are performed, the state of the action plan should be updated to **SUCCEEDED** with the execution summary in the **Efficacy indicators** and **Global efficacy** fields:

```
$ openstack optimize actionplan start 23a766da-77da-2361-bca1-
efeda124870d
```

The output is as follows:

```
+---------------------+------------------------------------------------------------+
| Field               | Value                                                      |
+---------------------+------------------------------------------------------------+
| UUID                | 23a766da-77da-2361-bca1-efeda124870d                       |
| Created At          | 2024-02-29T20:22:21+00:00                                  |
| Updated At          | 2024-01-29T20:55:54+00:00                                  |
| Deleted At          | None                                                       |
| Audit               | da23a982-deaa-3231-3120-237dea123cba                       |
| Strategy            | vm_workload_consolidation                                  |
| State               | SUCCEEDED                                                  |
| Efficacy indicators | - Description: The total number of enabled compute nodes.  |
|                     |   Name: compute_nodes_count                                |
|                     |   Unit: null                                               |
|                     |   Value: 3.0                                               |
|                     | - Description: The number of compute nodes to be released. |
|                     |   Name: released_compute_nodes_count                       |
|                     |   Unit: null                                               |
|                     |   Value: 1.0                                               |
|                     | - Description: The number of VM migrations to be performed.|
|                     |   Name: instance_migrations_count                          |
|                     |   Unit: null                                               |
|                     |   Value: 2.0                                               |
|                     |                                                            |
| Global efficacy     | Released_nodes_ratio: 33.33 %                              |
|                     |                                                            |
| Hostname            | cn02.os                                                    |
```

Figure 9.28 – Watcher optimization action plan start

By checking the Watcher dashboard, we can notice the change in the instance counts across the listed hypervisors where `cn01.os` becomes completely free. Watcher has determined that a given workload can run in two compute nodes instead of three, leaving one hypervisor machine's resources free, as shown:

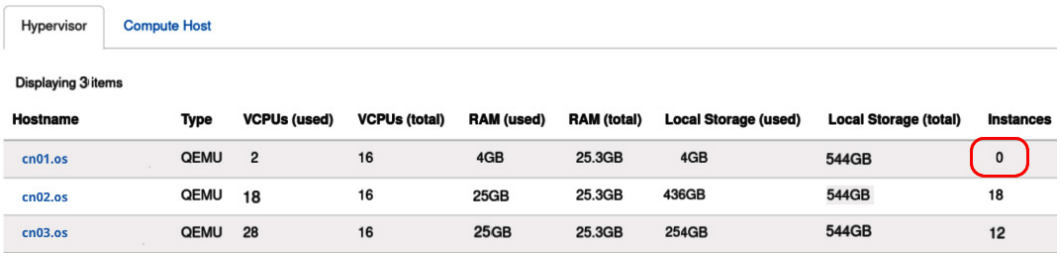


Figure 9.29 – The instance distribution state after optimization is applied

With this optimization achievement, cloud operators should get more free resources to route requests to for future workloads. Predefined goals are a great start to performing optimization exercises. Operators can define custom strategies and goals for different cases based on workload needs as well as the scheduling mechanism that has already been configured. From a big-picture perspective, a well-defined filtering and scheduling configuration is beneficial not only to tackle customization on compute resources and allocation but also to help the Watcher planner find the best optimization options. That will save operational overhead and a good amount of costs.

Summary

In this chapter, we brought our OpenStack setup to the next level by highlighting a few advanced settings that leverage its performance, such as the database. You should now understand the necessity for undergoing rigorous and effective testing of the cloud platform.

There is a learning curve in the art of benchmarking, which gives insight into all facets of the components running in the OpenStack environment, including system hardware and software resources. The chapter highlighted a tiny component, OSProfiler, that can be installed easily in OpenStack to trace and debug requests traveling across the OpenStack services. With a complete view of requests, operators can collect more data and generate a detailed service map that can be used for further performance analytics. The last part of the chapter drew upon best practices relating to resource optimization and automating recommendations when opportunities exist to reduce costs and improve performance. With Watcher, operators will not need to seek a third-party solution unless more functionalities are required. Benchmarking, profiling, and optimization practices should be performed periodically. Although the chapter did not cover how to automate those practices, creating dedicated pipelines for each benchmarking, profiling, and optimization process is highly recommended.

This chapter closes the second part of this book. In the next part, we will look at some recent and modern approaches that leverage the benefits of private and public clouds. In the next chapter, we will go through some common cloud hybrid design patterns where OpenStack continues to shine.

Part 3: Extending the OpenStack Cloud

The final part of the book will crystallize a trendy topic covering the adoption of a hybrid cloud setup. Using OpenStack as a private cloud environment joined with a public cloud provider, this part of the book will demonstrate how to leverage both cloud models to spread workloads in the most efficient and cost-saving ways. Empowered with microservices design patterns and containerization technology, this part will explore different tools and ways to run a workload based on Kubernetes across a running private cloud based on OpenStack and a public cloud based on AWS.

This part has the following chapters:

- *Chapter 10*, *OpenStack Hybrid Cloud – Design Patterns*
- *Chapter 11*, *A Hybrid Cloud Hyperscale Use Case – Scaling a Kubernetes Workload*

10
OpenStack Hybrid Cloud – Design Patterns

"In union there is strength."

– Aesop

As organizations continue to leverage cloud technologies, there has been a remarkable shift in the way cloud services are consumed through different cloud models. Recent studies and experiences have shown an increase in the need to adopt a mixed approach to using managed services across multiple vendors at the same time. Following the **multi-cloud** track has become a top trend where enterprises seek the best options to accommodate their workloads in the most efficient and optimized ways. In the previous parts of the book, we have demonstrated the position of OpenStack in cloud parlance. However, for many organizations, running only a private cloud environment might not be sufficient to satisfy all business requirements. Similarly, public cloud providers offer a wider range of service layers including **Infrastructure as a Service** (**IaaS**), **Platform as a Service** (**PaaS**), and **Software as a Service** (**SaaS**). However, some companies may not consume all services publicly for specific reasons. OpenStack's private cloud might be the answer to start a cloud journey but, for whatever reason, it might not be enough. The compromise found for several organizations is to adopt a hybrid approach where both public and private clouds are used. As we have a running private cloud based on OpenStack, this chapter will expand the OpenStack perimeter by going through the following topics:

- Exploring the hybrid cloud model
- Revisiting the public cloud model and multi-cloud strategy
- Constructing a hybrid cloud strategy with OpenStack
- Iterating through hybrid cloud design patterns

Hybrid cloud under scope

Cloud technologies have been immensely adopted for over a decade. When moving from the on-premises world, organizations seek different ways to increase agility by finding the best offerings based on IaaS, PaaS, and SaaS models. Besides the versatility of the service delivery models, the next terms that started emerging were the private and public clouds. Other emerging models have been adopted by several organizations when seeking the best cloud strategy. In IT parlance, we can find references to an organization being multi-cloud by using more than one cloud provider or by combining both public and private cloud services, which makes a **hybrid cloud environment**.

Defining the right cloud strategy depends on each organization's needs, which vary based on geographic presence, security, and budget requirements. Ultimately, not every workload is permitted to move to the public cloud environment due to data compliance control, for example. In other cases, an organization will pick a public cloud provider to be present in more than one geographic location that has available resources on demand and start serving the business immediately. Before getting to grips with the hybrid cloud context, let's look at the public cloud model.

Public cloud strategy

The rise of cloud usage has been observed more in the public cloud world. As *public* means *available over the internet*, organizations just needed to access a self-service portal to provision infrastructure on demand and enjoy elasticity and infinite scalability. Major cloud providers such as **Amazon Web Services** (**AWS**), **Microsoft Azure**, and **Google Cloud Platform** (**GCP**) keep extending the number of managed services in regular cycles. Public cloud consumers benefit from each new managed service that reduces operational overheads and consumes services with less effort and even with less required expertise. A presence around the world is another important deal for many organizations that are active globally. Most of the big public cloud providers have built dozens of data centers across several regions worldwide, making them an attractive option to support businesses in different locations. Hence, developers can focus on the business logic of the application. They can also run experimentation at a lower cost without spending an immense budget upfront. The public cloud has also solved the *agility mystery* in which startups and medium enterprises have been struggling to afford an infrastructure that could provide agility. Although the term *DevOps* existed even before the public cloud revolution, the lack of tools and, hence, the infrastructure that can leverage the whole agile software life cycle has been a major barrier. Before the introduction of the public cloud realm, the **Infrastructure as Code** (**IaC**) concept was just a theory.

Operation teams will not spend the same time and effort on traditional administrative tasks but, rather, will seek better performance metrics such as latency and response time. Infrastructure resources available from public cloud providers are offered without limits to consumers. However, the underlying resources are, at some point, still limited. Public cloud providers regularly estimate the resource utilization per region, per availability zone, and even per server rack. With local teams actively monitoring and extending the resource pools on demand, cloud consumers do not need to worry about limits as they can still request more when needed.

Today, most of the public cloud vendors offer different service delivery models including IaaS, PaaS, and SaaS. As shown in the following figure, moving from left to right, more layers fall under the responsibility of the cloud provider. Most cloud providers aim to offer the three service delivery models that can be found in major hyperscale public cloud providers (IaaS, PaaS, and SaaS), but not all private providers cover all three. As shown in the next figure, each service model defines the perimeter of responsibility shared between a cloud provider and cloud consumers:

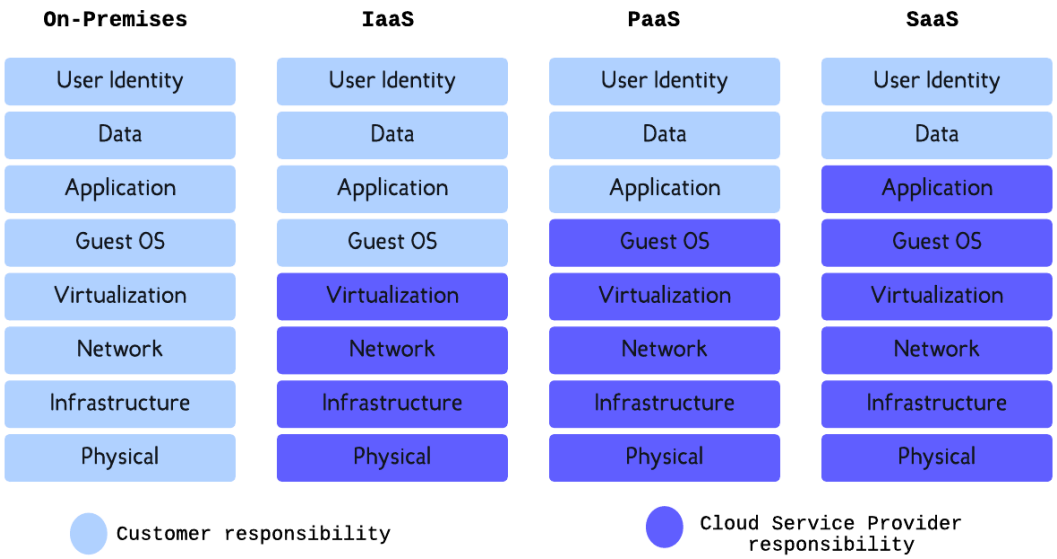


Figure 10.1 – The responsibility boundaries of cloud service models

In a traditional on-premises environment, an organization is fully responsible for managing all different aspects of the workload running in its infrastructure. By moving to a cloud environment, the burden of management and operation is reduced when moving from an IaaS to a SaaS model.

We should also add the innovation aspect when choosing a cloud service provider. With the rise of each new trending technology, public cloud vendors rush to wrap a managed service and make it available for consumption. Taking the example of the booming **Artificial Intelligence** (**AI**) realm, at the time of writing this edition, we find that most of the big public cloud players have harnessed notable initiatives for AI. AWS has brought its **Bedrock** cloud service to life as a generative AI-specific PaaS in the cloud. Microsoft Azure has quickly run to partner with OpenAI, a private AI research organization, and ended up bringing it under the umbrella of the Azure AI Service suite. GCP built its own AI PaaS platform named **Vertex AI**.

AI-managed services in the public cloud can take different names and choosing a cloud provider can be a challenging decision. The answer can be simple: *it really depends!*

That answers not only the given AI case but also, ultimately, each new similar-sounding service with each cloud provider. For that reason, organizations facing the *paradox of choices* should invest time and budget to learn about the new edge service, give it an experimental milestone, and conclude whether it fits with what they need or not.

A gazillion advantages can be noted from adopting a public cloud approach. On the other hand, after a few months and years of consuming public cloud services, organizations have started observing a few shortcomings. The adoption of the public cloud is based on a **pay-as-you-go** model; this is what has convinced enterprises that it is the right choice to reduce the costs of traditional infrastructure. However, if it is not carefully analyzed, it could be more expensive than planned. Whether a business is migrating to the cloud or is born in it, as reflected in the approaches to investment, building a business case and determining the **Return On Investment** (**ROI**) is a must-do practice that should be done at the start of the journey. A simple business case should look at the capital expenditure and cost when leveraging public cloud resources per workload. For example, an application can run on top of one or several virtual machines. A virtual machine that assumingly runs 100% of the time can keep a linear cost curve. If this is not carefully tagged and tracked from the cost point of view, operation teams can easily lose track of where the budget is spent. **Hyperscaler** providers offer transparent billing options and more granular cost breakdown views to help customers understand their usage. However, in most cases, optimizing workload spending would require some changes to its architecture and infrastructure components. The fear of such changes breaking things in production could discourage operators from taking that experimental cost optimization step. Cost in the public cloud world can be a major disadvantage if it is not estimated in advance. For that reason, a **Financial Operations** (**FinOps**) discipline has been created that aims to make financial accountability to the cloud a collective matter between IT, finance, and business teams. To read more about FinOps, you can check the official FinOps foundation website, which can be found at `https://www.finops.org/`.

The next issue with public cloud usage is security and compliance concerns. When public cloud providers started exposing managed services, security teams brought forth questions about where and how the data was being transferred and stored. Although public providers have clarified their compliance certifications that prove their underlying infrastructure security, several organizations are still not comfortable spreading their data in locations other than their own data centers. We know that some compliance controls, such as **Sarbanes-Oxley** (**SOX**), **International Standards Organization (ISO**), and so on, are well supported by public cloud providers. However, organizations that operate around the globe should have a hosting infrastructure compliant with each region that is running its business. Taking the example of the **General Data Protection Regulation** (**GDPR**) compliance standard within the **European Union** (**EU**), cloud providers such as AWS had to take rigorous measurements to protect data for transfer from the EU to non-EU countries and regions. Hence, any region other than the EU must also comply with it.

From a security standpoint, cloud providers define security accountability as a shared model between the provider and the consumer. As highlighted in the following diagram, AWS defines the perimeter of security responsibilities *in* and *of* the cloud in its **shared responsibility model**:

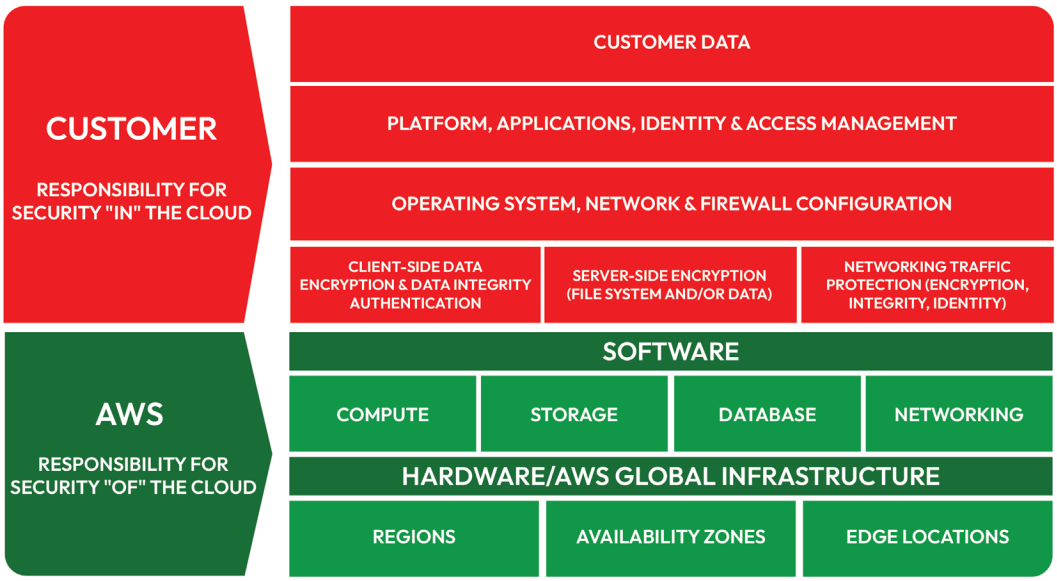


Figure 10.2 – The AWS shared responsibility model

The orange layer is where the provider defines its responsibility to secure the underlying infrastructure and make sure it is compliant with regional regulations. That covers not only security but also resiliency and service availability, which varies from one service to another. To read more about the *AWS Risk and Compliance* whitepaper, refer to the AWS Compliance web page at `https://aws.amazon.com/compliance/`.

The next concerning point on the list is the lack of full control over the cloud infrastructure. At first glance, the preceding diagram might seem counterintuitive. The fact that organizations are looking for a public cloud offering is mainly due to running faster time to market without the need to get their hands into the hardware details or underlying networking devices. That can be a fair reason to adopt the public cloud, but what about when we need the customization of the underlying hardware? As that is managed only by the vendor, consumers will not have control over it. That might sound odd from the first reflection, but compared to traditional on-premises models, sometimes application teams ask for certain hardware combinations with granular specifications that cannot be found in the public cloud. Network teams, in some edge cases, could be asked to build a network map that is not convenient in the public cloud world. Coming from the on-premises world, cloud consumers might have some expectations of having a certain degree of control of the underlying infrastructure that is not feasible at the beginning. That can turn into accepting such limitations at the cost of architecting a complex network setup. On the other hand, some cloud providers have responded to many customers' requests and provided certain ways of controlling how hardware can be provisioned to run specific workloads. For example, AWS offers a **dedicated instances** feature that allows for running a bunch of EC2 instances on hardware that is dedicated to a single AWS environment and isolates them from sharing the underlying physical machines with other instances belonging to other AWS environments.

Although cloud providers are aware of such limits, which is the purpose of the **managed services model**, enterprises keep seeking the maximum number of ways of having control over the cloud to achieve the best results.

The last public cloud drawback point is the vendor lock-in dilemma. Protection from lock-in situations is a major requirement for most enterprises that seek to be *infrastructure-agnostic* before being *cloud-agnostic*. Moving freely between environments and platforms is a luxury driver for many businesses to stay economical with resource usage and respond to development requirements when needed. Lock-in can take different forms. For example, let's say that an organization wishes to move its workload running on a virtual machine between different cloud providers or on-premises. That would require a longer time for planning and testing before doing such a move, such as building a private connection between the cloud source and the environment destination(s), building the application artifact, deploying it in the target environment, and testing it. The good news, thanks to some amazing open source initiatives, is that containerization technology has been a rockstar that has enabled standardized application deployment to a certain degree. Hence, moving between environments is a matter of connecting them with a shared container registry. The other vendor lock-in form is the way of managing the infrastructure itself. We have highlighted previously how the IaC concept started shining only when the public cloud went into action. As soon as more services and resources became overwhelming to manage, public cloud providers built the right managed service to codify the infrastructure. We can find, for example, **CloudFormation** in AWS, **Azure Deployment Manager** in Azure, and **Cloud Deployment Manager** in GCP. If you land in AWS and start writing code templates for your AWS resources, it won't be a viable option if you later plan to use the same code abstract in GCP or Azure. Being cloud-agnostic when it comes to IaC is almost a religious matter for many enterprises. In many cases, those enterprises skip learning any of those resource management tools and stick to cloud-agnostic ones such as **Terraform** and **Pulumi**.

As this section summarizes why the public cloud matters and what can be feared from adopting it, the next section will show in the same way why the hybrid cloud model should be considered.

Hybrid cloud strategy

Based on several *state of the cloud* reports, numbers show the continued growth of public cloud services adoption. The *RightScale State of the Cloud Report* from *Flexera* (more recently known as *Flexera State of the Cloud Report*) has been following the state of the art of cloud adoption. For more than 12 years, Flexera has been conducting surveys of corporate cloud users from small to large enterprises. A few interesting numbers are highlighted based on the survey report of 2019 (`https://resources.flexera.com/web/media/documents/rightscale-2019-state-of-the-cloud-report-from-flexera.pdf`), showing that organizations run more than 38% of their workloads in the public cloud. Meanwhile, more PaaS services are being adopted and keep skyrocketing with 50% growth compared to the previous year. The same report shows that 38% of

companies see the public cloud as a top priority for their IT infrastructure compared to the previous year's 29%. Those numbers seem to be promoting public cloud adoption overall. However, we might be surprised by some findings in the same State of the Cloud report where enterprises expressed their will to combine public and private clouds for their IT usage. As per the report in 2018, 51% of companies were looking for a hybrid cloud strategy, which grew in the following year to reach 58%. This pattern might match with another *State of Cloud Adoption Report* conducted by Forbes in 2017 (`https://www.forbes.com/sites/louiscolumbus/2017/04/23/2017-state-of-cloud-adoption-and-security/?sh=69d3eacd1848`) showing that hybrid cloud adoption has been increasing by about three times compared to the previous years (increasing from 19% to 57%). The following figure illustrates the growth of hybrid model adoption between 2015 and 2016, taken from the Forbes 2017 report. We can clearly see that both public and private cloud adoption decreased in favor of a hybrid model:

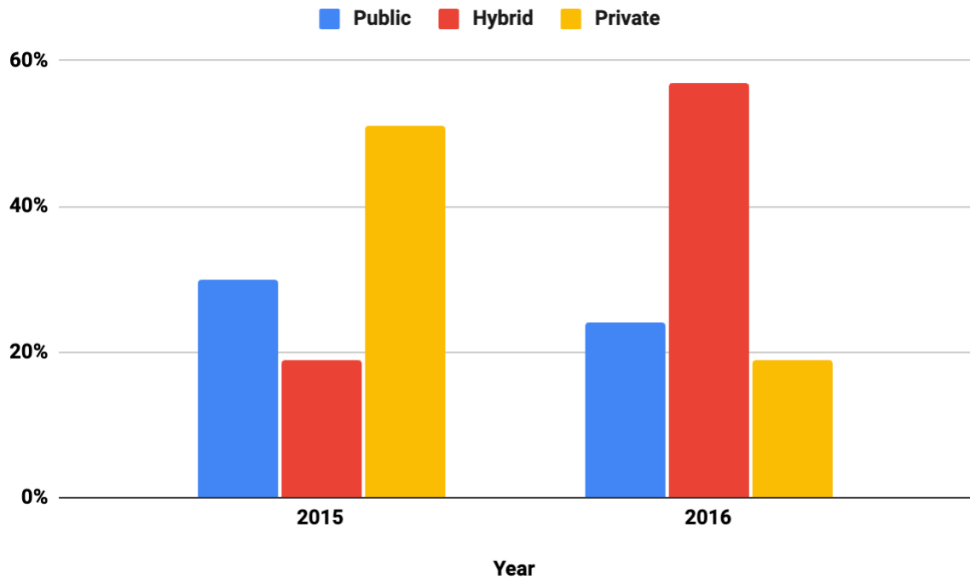


Figure 10.3 – Types of cloud models used based on the Forbes State of Cloud Adoption Report 2017 report

Although those statistics are from seven years ago, the 2024 *Flexera State of the Cloud Report* (`https://www.flexera.com/about-us/press-center/flexera-2024-state-of-the-cloud-managing-spending-top-challenge`) demonstrates that hybrid cloud adoption remains the most preferred option for several organizations. By gaining more cloud experience and with more cloud lessons, organizations have decided to move to a multi-cloud pattern (either with different public or private providers or by combining both). With each public cloud move, a company combines the public world with the private cloud. As shown in the latest report, about 73% of surveyed cloud decision-makers and cloud enthusiasts consider a hybrid cloud strategy for their IT infrastructure:

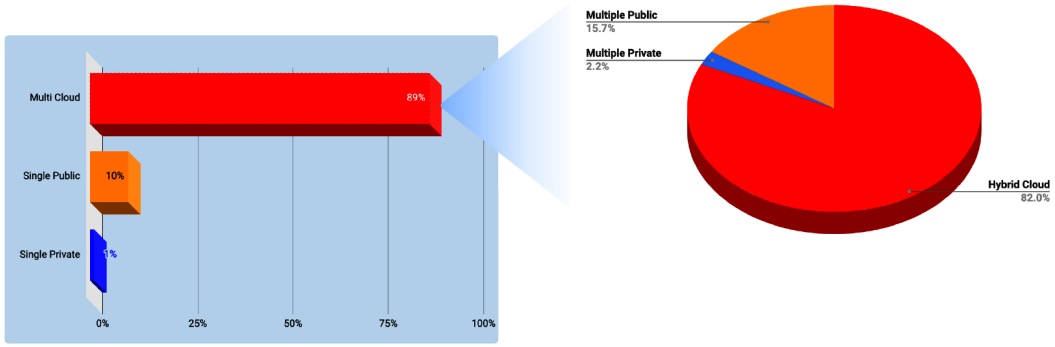


Figure 10.4 – The results of a multi-cloud adoption survey based on the Flexera 2024 State of the Cloud Report

There are various reasons that explain the expansion of hybrid cloud model usage. The best definition to get started with is how Gartner defines a hybrid cloud environment: *"Hybrid cloud computing refers to policy-based and coordinated service provisioning, use, and management across a mixture of internal and external cloud services"* (`https://www.gartner.com/en/information-technology/glossary/hybrid-cloud-computing`). In a nutshell, the combination of more than one public and private cloud addresses the shortcomings of using only public or private clouds. We can see that hybrid models can satisfy what organizations could not find in the public or private model. Let's iterate through some advantages brought by the adoption of the hybrid cloud model:

- **Cost efficiency**: With a steady utilization of private cloud resources, occasional peaks can be outsourced to the public cloud, thus keeping the cost to a minimum. Proofs of concept and testing environments can turn out to be more expensive in a public cloud environment if not monitored properly. In some cases, developers can spawn large EC2 instances, and forgetting to terminate them after testing can be ultra expensive. Such scenarios are very common if cost control policies are not set properly. In this case, running in a private cloud can be less expensive and can easily be detected as resources are not infinite compared to resource pools in the public cloud. For workloads running on servers with periodic licensing renewal and maintenance efforts, the public cloud can be the right place to host those workloads and save maintenance costs and management overhead compared to the private cloud.
- **Governance and compliance**: Data residency can be more flexible to meet regularity requirements for each workload. Not all data can be hosted in the public cloud, and hence, a private infrastructure environment can satisfy such restrictions without affecting the application's business continuity.
- **Customizability and control**: Not having a hand in the underlying infrastructure can be a blocker for some workloads. Public clouds do not offer a direct way to dictate low-level configurations, including compute hardware and networking devices. Such customization can be adjusted by using private resources alongside public environments.

- **Lock-in-free**: A multi-cloud strategy is becoming an important aspect for some organizations to stay flexible with the deployment of their workloads. Being vendor lock-in free in a hybrid model allows for moving workloads between private and public infrastructures without being fully committed to a specific vendor. Containerization has boosted the adoption of hybrid clouds due to the portability criteria allowing applications to run across multiple cloud providers and move between environments with the least amount of time and effort.
- **Empowered resiliency**: Running workloads across dispersed infrastructures increases the level of redundancy. Some critical applications can run in both private and public environments. That would incorporate a stronger disaster recovery strategy if a private cloud availability zone were out of service; the public environment could come to the rescue by just pointing to its endpoint in a short period of time. Workload components can run in a *dispersed* mode in both public and private environments. Such a case requires a workload component, for example, to run in the private environment and interoperate with other components via a secured connection in the public cloud to leverage a managed service.
- **Increased scalability**: **Cloud bursting** is a key element of a hybrid architecture model. Having more options to deploy a workload with the combination of the private and public worlds increases agility. Organizations use cloud bursting to scale faster and respond to spikes when demand exceeds the private cloud capacity. Cloud bursting use cases will be explored in more detail later in this chapter.

Recently, hyperscaler providers have been following the hybrid cloud adoption trend and offering hybrid cloud platforms as part of their managed cloud services. For example, AWS has launched the **AWS Outposts** service enabling cloud users to run AWS services locally within the organization's perimeter. To read more about the AWS Outposts case, refer to the AWS official document at `https://docs.aws.amazon.com/outposts/latest/userguide/what-is-outposts.html`.

Similarly, we can find the **Azure Stack** service in Azure and the **Google Anthos** service in GCP.

Enterprises have realized from their public cloud experiences that a public cloud is not always the best bus to ride. The rise of private cloud capabilities such as OpenStack makes it possible to be present in both public and private clouds. Organizations adopting OpenStack as a private cloud solution find themselves in a great position to maximize the benefits of using both worlds. The modular architecture and its rich APIs make OpenStack a great fit to work in conjunction with the public cloud environment, which we will look at in the following section.

OpenStack meeting the public world

Architecting for hybrid cloud models can be a great deal for several organizations seeking the aforementioned hybrid cloud benefits. If you are building a private cloud based on OpenStack either for internal usage or for potential customers, cloud operators can extend the infrastructure perimeter to combine with public cloud services to create valuable outcomes. In a hybrid model, it is a good method to create a centralized platform that manages both the private and public clouds acting as a

single pane of glass. The platform does not need to add complexity to talk to different cloud providers' APIs but simply an administration dashboard with proxying capabilities to route requests that span different environments. With an existing OpenStack environment, cloud operators can establish secured connections in high bandwidths between the OpenStack private cloud and each of the public cloud endpoints, as shown in the next diagram:

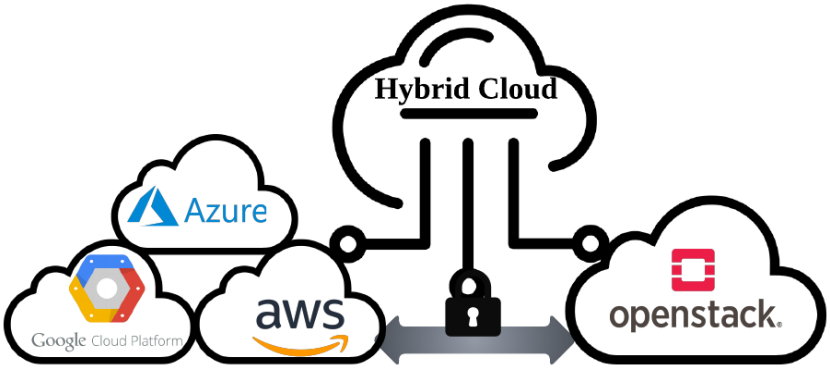


Figure 10.5 – A generic hybrid cloud architecture

Combining both the OpenStack private cloud and public cloud vendors needs to satisfy certain conditions to make a complete hybrid cloud picture. In this case, a business workload should be designed to run in an OpenStack environment, in the public cloud, or both. The other important condition is the readiness for data mobility between private and public clouds. That would require a good understanding of when data is allowed to be stored in public or private infrastructure. Some other cases would require, for example, having short-lived data in the public cloud while keeping long-lived data in a private cloud.

Another essential aspect to be considered when running workloads in a hybrid setup is the compatibility and variation between the different types of cloud. You might run a workload between two or more different clouds supporting multiple APIs of the same kind with the same technology stack. That can be less complex to run with easier implementation. The other variation is when different clouds do not support the same kind of APIs with different types of components, such as database engines or types of hypervisors. In this case, more tweaking is required to increase the degree of complexity. The good news is that OpenStack has been designed to leverage some of the public cloud capabilities. The secret sauce is its rich API exposure. As a private cloud software, OpenStack is considered a good choice that can power the network of private and public clouds. OpenStack is leveraged to manage resources for other public cloud providers such as AWS. For example, OpenStack supports a few drivers for AWS such as Nova for EC2, Neutron for VPC, and Cinder driver for EBS in AWS. Those drivers come under the **OpenStack-Omni** project, which can be found at `https://opendev.org/x/omni`. Although this initiative is still moving at a slow pace, the OpenStack community is looking at integrating more capabilities to the OpenStack API to manage hybrid and multi-cloud environments such as GCP and Azure.

Ideal cloud hybrid implementations leverage ways to centrally control different resources in public, private, and on-premises through a unified **Cloud Management Platform** (**CMP**). As shown in the next figure, providing an extra layer or tools for hybrid cloud management helps centralize the administration of workloads and data across any kind of infrastructure:

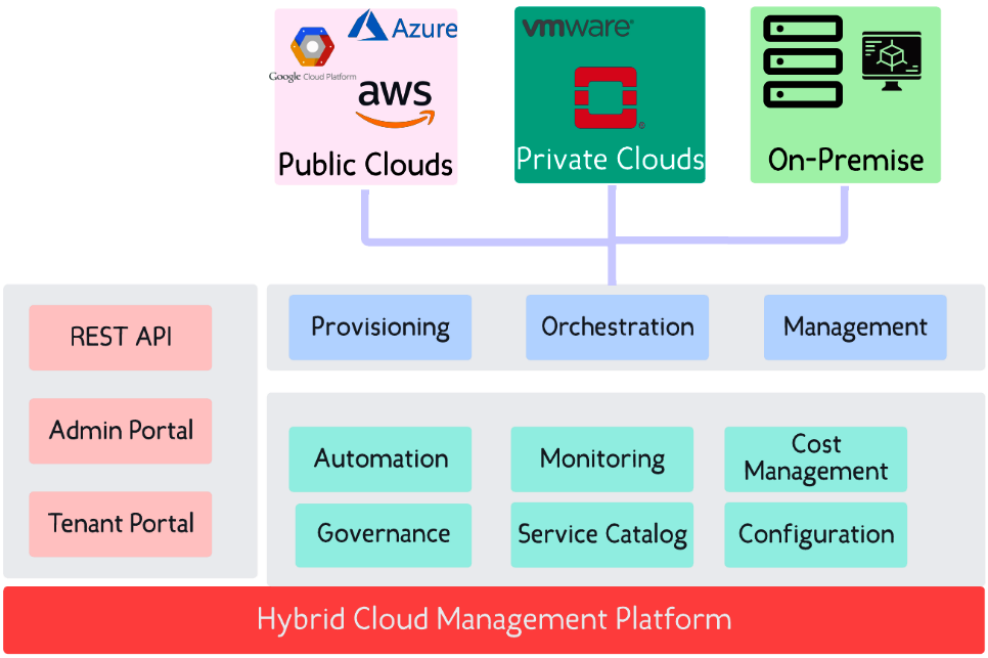


Figure 10.6 – The capabilities of a CMP

Today, more organizations are learning to develop their own CMPs for their own business needs or for public consumption. In the market landscape, Flexera, CloudBolt, OVHcloud, and Nutanix (to name a few) provide a unified platform to administer resources in both public and private clouds. A CMP for hybrid architectures is empowered by orchestration tools to operate across cloud environments. A hybrid management layer can be provided by a service catalog and reusable blueprints to provision infrastructure across all linked cloud endpoints with the maximum degree of automation.

Cloud providers realize the different needs of cloud customers to design more services that facilitate a hybrid cloud deployment. Such services can be part of a CMP or run on their own to operate a workload across private and public clouds, as we will explore in the next section.

Enabling the hybrid cloud

Today, we can find a range of managed services by giant public providers that facilitate building hybrid architectures. AWS has been one step ahead by offering multiple cloud services to satisfy customers' wishes with a hybrid model. For example, AWS Outposts leverages AWS resources by

using the same AWS APIs and tools but provisioned and located at the organization's premises. On top of Outposts, cloud users are offered the opportunity to run **Relational Database Service** (**RDS**) managed databases to take advantage of fully managed databases that scale for MySQL, Microsoft SQL Server, and PostgreSQL. To read more about RDS Outposts, refer to the following link: `https://aws.amazon.com/rds/outposts/`

Some amazing hybrid services have been unleashed for containers, including **Elastic Container Services Anywhere** (**ECS Anywhere**), which you can read about at `https://aws.amazon.com/blogs/containers/introducing-amazon-ecs-anywhere/`, and **Elastic Kubernetes Service Anywhere** (**EKS Anywhere**), which you can read about at `https://aws.amazon.com/eks/eks-anywhere/`. Both hybrid container services enable cloud users to run containerized workloads on-premises by keeping the benefits of scalability and orchestration toolsets. Hybrid storage solutions have become necessary for several organizations to perform backup, restoration, and disaster recovery between on-premises and public environments. AWS provides the **AWS Storage Gateway** service for hybrid cloud storage solutions. More information about the AWS Storage Gateway service can be found at `https://aws.amazon.com/hybrid/services/`. Other use cases would require an accelerated way to transfer data between private and public clouds. AWS has introduced the **AWS Datasync** service (`https://aws.amazon.com/datasync/`) as a hybrid storage solution.

Cloud bursting

Choosing where to run workloads depends mainly on customer business and governance requirements. Some organizations might dictate in which region and cloud they wish to host their applications. Others might prefer to deploy their services on one cloud or the other, but data (including databases or big data) must reside in a private cloud environment or even on-premises bare metal. Anything else that does not include data can run on the public cloud. With this variety of requirements, cloud operators should find a hybrid model that can fit each workload's needs. With the rich set of APIs of OpenStack, integration with the automation tools and external APIs of other vendors makes it easy to run as a hybrid with the least risk of vendor lock-in. By gaining such flexibility, growing the business application is no longer a concern to run in any public or private cloud environment. For example, let's say that an organization develops a service that gets packed on a container image. The image is deployed in a private OpenStack cloud environment and, quickly, a global presence of the application becomes a business requirement with a larger audience. That is where bursting comes into play. The workload would burst into a public cloud by using its resources in the nearest requested region. With the right design, cloud architects should be able to make any application live in both worlds and take advantage of the hybrid model capabilities without being locked into one model. Cloud bursting is a very common use case of a hybrid cloud, and is even considered a hybrid feature in other literature. As shown in the following diagram, cloud bursting leverages the utilization of the hybrid cloud infrastructures to more rapidly process sudden increases in requests and spikes in traffic for an application:

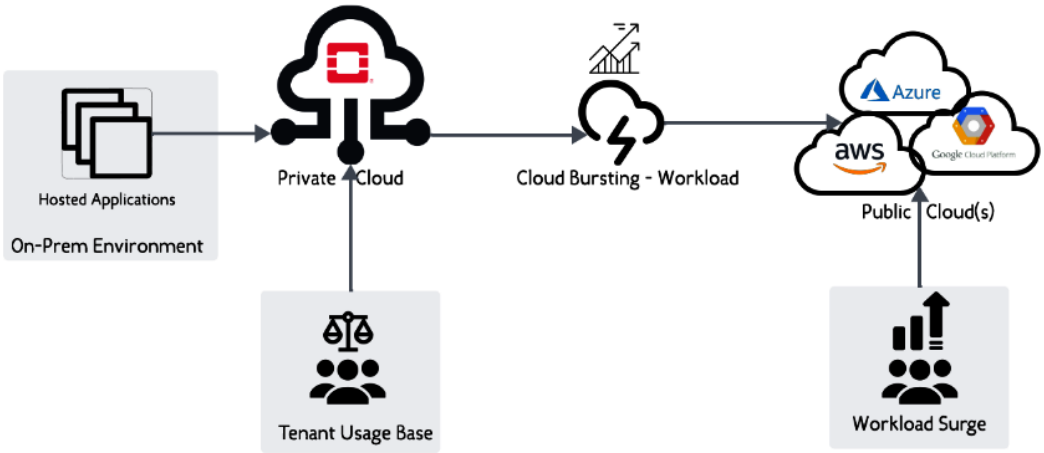


Figure 10.7 – Cloud bursting for the hybrid cloud use case

Ultimately, a company predicts its private cloud infrastructure utilization and responds to additional load by spreading it to public cloud resources. Cloud bursting is essentially a dynamic deployment of workloads to increase elasticity and scalability in peak times once the private cloud resource demand becomes overwhelming. The common pattern of cloud bursting is that companies rely primarily on their private cloud resources and enable bursting only during traffic spikes to accommodate the additional load. Once the sudden surge of the workload spike is over, public cloud resources are terminated and the application keeps running only in the private cloud environment.

Security in the hybrid cloud

Security *in* and *of* the cloud are two aspects that do not differ between the private and public clouds. In a hybrid model, cloud operators would review what is offered from a security standpoint. To a certain extent, security in a hybrid model can be enforced more when leveraging the private part, but additional threats can be observed with insecure API endpoints, data leaks, or missing encryption at rest and in transit. In the OpenStack world, cloud operators should think of the security *of* the private cloud, whereas security *in* the cloud is the task of the user. Following a security discipline would not differ from pure private or public clouds when dealing with a hybrid cloud environment. Starting with the principle of **zero-trust**, cloud architects should take security as a core principle at the earliest stage when designing a hybrid environment, in the same way as building a private cloud. In a hybrid setup, more components are involved in the security configuration. Based on a zero-trust approach, any entity of the architecture is considered a threat by default, including user access, workloads, data, network devices, and traffic. If you have ensured that your private cloud based on OpenStack is following the same approach, you would just need to combine the security boundaries of the cloud provider and shared responsibilities. The good news is that most hyperscaler providers have defined a shared responsibility model. Boundary responsibilities should be clarified at the start of the hybrid

cloud journey. For example, if a user is looking at a SaaS solution that is not supported by the private OpenStack cloud, the security responsibility is shifted to the target public cloud provider. Several entities are engaged when defining a shared responsibility based on a hybrid cloud setup, including hardware, storage, encryption, physical network, guest operating system, network and platform security, data location, user authentication, and authorization, to name a few.

In the following sections, we will take an iterative approach to demonstrate different entities involved in establishing security patterns around a workload running in a hybrid cloud environment. That includes the use case of scaling an application running on Kubernetes between an OpenStack environment and a public cloud.

Governance first

Data protection is one of the major motivators that encourage organizations to adopt a hybrid environment setup. Businesses and decision-makers define the rules of data privacy requirements and cloud users are responsible for adhering to them. Cloud providers should comply with defined rules of data protection, which will vary from one region to another. Taking the example of GDPR rules, the hybrid cloud should provide the insurance of data protection within the EU, including data privacy, residency, and retention. In a hybrid world, a mix of a private cloud based on OpenStack and a public cloud should be aligned with the following shortened list of requirements:

- Transparently define an extended version of the shared responsibility model.
- Document different aspects of security configuration within the hybrid environment. This includes how private and public clouds are connected and how data is transmitted and stored.
- Provide ways for users to access and leverage best practices using access policies, the least privilege principle, and encryption.
- Define data life cycle standards and measurements of data classification.

Ultimately, workloads hosted in a hybrid cloud should not stick to a single compliance requirement but rather to several due to the mixed nature of such a public and private combination. Such a process can be error-prone if not all governance measures are checked before starting to receive workloads. For this matter, it is essential to spend a good time establishing a well-secured platform between both private and public clouds. A containerized application can be easier to deploy across private and public clouds. We have covered the maturity of the OpenStack ecosystem when dealing with containers in previous chapters. For example, spreading a containerized application across an OpenStack private cloud and a public cloud environment should prompt the following governance and compliance considerations:

- How and where will my Kubernetes cluster be running in the short and long term?
- Will any data be stored in the private and public clouds?
- How does the hybrid cloud control access to my Kubernetes cluster and hence the applications running on top of it?

- Does the hybrid cloud provide a single pane of glass for monitoring and logging my application?
- Is there a standard incident management process to report security issues when data leakage or unauthorized access happens?

Compliance and governance in the hybrid world can be more complex to establish as each workload would present different compliance requirements. Standardizing the rules defined for the cloud infrastructure can reduce such complexity but requires a deep understanding of analyzing the perimeter of responsibility of each public cloud provider. From a private cloud perspective running OpenStack, you should have already a greater view of all pieces of the security puzzle when designing the private cloud.

Network security

Networking is usually the next discussed point for a cloud consumer to deploy a workload in the cloud of any type. The public cloud provides gazillions of ways to design networks for workloads and it is down to the cloud user to define the network perimeter. If the application is made to be publicly accessible, cloud users would leverage internet gateways or appliances to expose the application in its virtual network. Hybrid environments will be required to provide the maximum degree of flexibility for networking options. As we have seen in the OpenStack world, the underlying infrastructure includes not only the physical routers and switches but also an overlay network layer on top of the physical network infrastructure (software-defined networks). With both network layers (physical and overlay) being virtual, private cloud operators should consider ways of segmenting networks while keeping full control. The underlying network security layer that is not visible to cloud users must exist before hosting any workload. That can be achieved by leveraging firewalls or by putting web application firewalls in place to protect against DDoS threats. Similarly, public cloud providers ensure (based on the shared responsibility model) a certain level of network security, including hardware and traffic management. Traffic circulating within the public provider backbone is considered trusted, but it is down to the cloud users to define how the application will be isolated or made public. Either way, network services are provided to control the type of traffic within its network boundary or in wider exposure. If you are scaling an application between a private and public cloud, a variety of network security mechanisms must exist, which can be summarized as follows:

- Secure traffic reaching any cloud-type endpoint that supports the **Transport Layer Security** (**TLS**) protocol (v2 and v3).
- Make sure application containers are deployed and encrypted at rest.
- Provide secure mechanisms to protect applications publicly accessible via load balancers.
- Enforce application security by leveraging **Web Application Firewall** (**WAF**) solutions that can protect applications across several cloud environments.

- Implement a layered network architecture by segmenting an application network to run in different subnets for each layer. Dispersed environments (public and private) can be configured to use a private address space (RFC 1918). That way, different parts of the application can still reach each other while being contained in isolated networks, reducing the risk of public exposure.
- Build private connections between different cloud environments. This is an extension of the previous point whereby an application can reach any cloud in the hybrid environment through a private connection without traversing the internet. Taking the example of a Kubernetes cluster, pods in an OpenStack environment can reach other pods of the same cluster context in AWS via a private connection such as **DirectConnect** or a **VPN**. Cloud operators would need to define the different routes to reach each of the cloud endpoints so the application components' interconnectivity stays private even across different cloud environments.

Hybrid clouds can be designed to use different types of connections and network implementations. Looking at the network security aspect requires solid preparation for securing endpoints for both private and public clouds. The private part would require more hardening as a cloud operator has a full hand on the network infrastructure. In our case, securing the API endpoints in each service in the OpenStack private cloud is a must. The OpenStack community provides an updated version of a security guide upon each new release that is worth visiting. The API endpoint security guide can be found at `https://docs.openstack.org/security-guide/api-endpoints/api-endpoint-configuration-recommendations.html`.

Hardening operation security

The nature of the dispersed architecture of an application across different cloud infrastructures can be a challenge when it comes to its operation. Taking advantage of cloud bursting in a hybrid setup comes with the price of the definition of operation security responsibilities. By inheriting the practices of the **DevSecOps** mantra, the operation complexity in a hybrid environment can be much more simplified. We have dedicated the second chapter in this edition to illustrating how DevSecOps is an essential style to automate security in the operation of an OpenStack infrastructure. We will just need to apply the same approach when dealing with workloads that are spread across several infrastructures. The use case of an application running in a Kubernetes cluster in a hybrid cloud setup would require the following additional checks to operate in the most secure way:

- Use CI/CD pipelines to automate image scans of vulnerabilities.
- Dry run tests to detect and alert of any potential security gaps.
- Use a container image registry that is only accessible privately.
- Make the image registry available to public or private cloud endpoints only via private connections.
- Harden and automate the creation of container images in regular cycles and share only stable and well-tested versions with the rest of the cloud endpoints.
- Deploy application container images from one single CI/CD pipeline.

- Set up a logging pipeline to collect logs across different infrastructures. This can be achieved in a centralized way by pulling network flows, application logs, and firewall logs for potential issues analysis.
- Configure monitoring and metrics to report centrally on security issues for each cloud infrastructure endpoint.

Security in a hybrid model can be a lot more complex since having data flowing in all different infrastructures requires additional controls. Data handling should be carefully performed as improper data movement can raise major data leakages that are not easy to recover. The other challenge of hybrid implementation is the complexity of access management. Users will need to be authenticated and authorized in a flexible way that grants access across multiple infrastructures. Using **Identity and Access Management** (**IAM**) coupled with single sign-on tools can help to remediate access complexity challenges.

Summary

This chapter exposed the hybrid cloud novelty. Beyond the OpenStack private cloud capacity, there is always an opportunity to reach public cloud environments and construct a hybrid environment, as we learned in this chapter. The latter model keeps expanding as a new cloud style as enterprises have felt the necessity to have combined cloud modes to enjoy each type of cloud's benefits. We learned that recent cloud experiences have shown that a multi-cloud strategy is still not enough to fill cost gaps in addition to compliance concerns. The increase of the hybrid model dominance has been accompanied by more additional OpenStack private cloud implementations. Gaining skills to build and manage an OpenStack private cloud is one of the cornerstones to moving to a hybrid model, which is why we focused on this in this chapter. Furthermore, we learned that OpenStack provides various cloud capabilities and, thanks to its rich APIs, our private cloud can integrate well into a hybrid cloud model.

In the next chapter, we will demonstrate how building a private cloud environment based on OpenStack will ensure a successful hybrid cloud experience in which OpenStack keeps shining.

11

A Hybrid Cloud Hyperscale Use Case – Scaling a Kubernetes Workload

"There are not more than five cardinal tastes, yet combinations of them yield more flavors than can ever be tasted."

– Sun Tzu, The Art of War

In the previous chapter, we briefly explored hybrid cloud design patterns and how hybrid deployment has become a cloud trend for several organizations. The hybrid cloud model reflects a complex mix of services that work in tandem to satisfy different workload requirements. When you join an OpenStack private environment, a hybrid model can be more challenging to set up due to the complexity of the dynamic mix of resources spread across private and public environments. Building hybrid cloud networks is the first challenge that an organization would face, which requires stable, dedicated, and consistent connectivity between both public and private clouds. Moreover, applications that can run on bare metal, virtual machines, or containers require a way to ensure configuration consistency between different cloud environments. Several implementation challenges covered in the previous chapter still present a blocker for many organizations from enjoying the benefits of hybrid clouds. The good news is that with the evolution of **application architecture patterns**, **containerization** technology has shifted how and where applications can be deployed. If we are looking for application migration, cloud bursting, or running for a backup and disaster recovery strategy, adopting containerization technology is the way to go. In this chapter, we will put into practice what was covered in the previous chapter and uncover the power of containerization in a hybrid cloud setup by covering the following topics:

- Designing hybrid cloud networking between OpenStack private cloud and a public cloud environment such as AWS
- Revisiting briefly the main concepts of Kubernetes (also referred to as K8s) to prepare for a cloud-agnostic workload model across OpenStack and AWS clouds

- Exploring different approaches to implementing a hybrid cloud setup
- Deploying Kubernetes across an OpenStack private cloud and AWS public cloud
- Learning orchestration tooling for hybrid cloud deployment using Juju
- Connecting Kubernetes clusters across private and public clouds by federating its control plane and running a hybrid deployment

Qualifying a hybrid architecture

In this section, we will explore the main motivation for adopting hybrid cloud implementation. We will briefly illustrate how containerization helps achieve most of the hybrid model benefits and enables companies to move between cloud environments most efficiently. The networking and security of connected cloud environments is another important pillar that must be looked at carefully when adopting the hybrid approach.

Adopting a cloud-agnostic stack

Being cloud-agnostic is a key motivator for several organizations that seek the best cloud experience when being onboarded into a cloud journey. Thanks to the **pay-as-you-go** model, organizations can use cloud resources and pay only for what they use. However, some business workloads might require changes, and switching or expanding to another cloud provider might become necessary. Enjoying free-pass movement between cloud providers might not sound straightforward at first. This is due to the technology that a cloud provider offers. A workload naturally becomes locked into more dependencies that run on a specific cloud provider. Moving or expanding to another cloud provider can be an expensive, complex, and error-prone process. For that reason, cloud agnosticism should be addressed in advance, starting with the workload architecture itself. The best example of this is the emergent **microservices architecture**. One of the main benefits of opting for a microservice design is that it breaks an application into smaller and independent modules, making it easy to test and deploy them separately without the need to rely on a specific technology stack. Another benefit of a microservice design is the infrastructure that will run it. That is where containerization comes into the picture. Containers come with much more efficient compute utilization and time optimization compared to virtual machines. That makes the microservice and containerization concepts an ideal pair to adopt cloud agnosticism. Hence, the hybrid cloud model is much easier to implement, with some additions. As illustrated in the following diagram, those additions include a container orchestration layer and automated deployment, making the cloud-agnostic model almost complete:

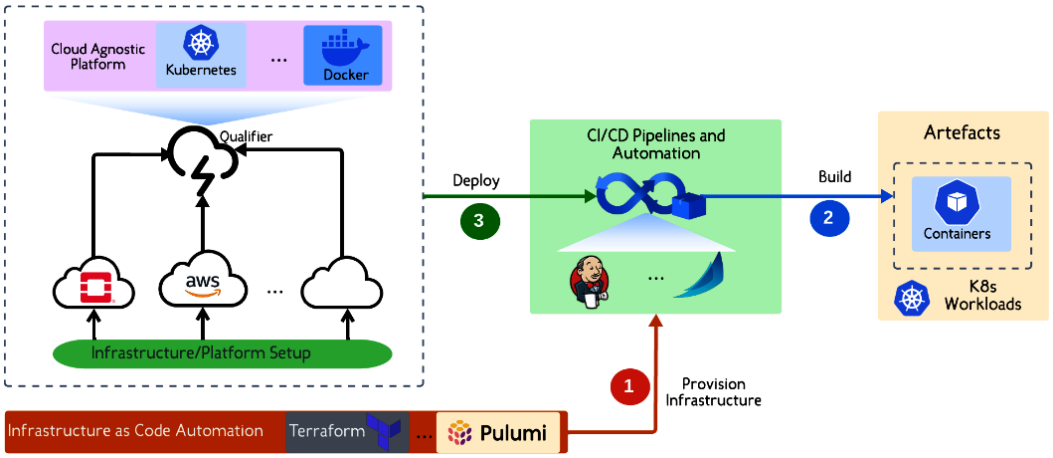


Figure 11.1 – The cloud-agnostic architecture of a hybrid model

Container orchestration engines manage the life cycle of the workloads with almost no administration need. Most of the famous container orchestration engines such as Docker and Kubernetes are considered part of the cloud-agnostic container orchestration platform family. While container orchestration brings more container management automation on its own, it would not be nearly as valuable without the **Infrastructure as Code** (**IaC**) concept. Each cloud provider offers its own infrastructure templating mechanism (**CloudFormation** for AWS, **Cloud Deployment Manager** for Google, **Deployment Manager** for Azure, and **Heat** for OpenStack). The vendor-specific syntax of each service presents a challenge as switching between cloud deployment environments becomes more complex. The solution is obviously to take a cloud-agnostic approach by employing third-party tools that allow us to deploy the same workload infrastructure in more than one cloud environment, using the same code to abstract the infrastructure layer. These tools became available only after the IaC concept had emerged as a DevOps principle. **Terraform** from Hashicorp has become the most commonly used cloud-agnostic tool for infrastructure management. Today, more open source projects have been inspired by Terraform, and recently, the **Pulumi** tool has appeared on the DevOps horizon. In addition to managing infrastructure using code, Pulumi allows the description of the infrastructure in any o the most popular programming languages such as Python, Java, Go, .NET, and Typescript.

> **Important note**
>
> On August 10, 2023, Hashicorp announced that Terraform is no longer under the Mozilla Public License v2.0. Pulumi has emerged as a successor of Terraform in the open source world and has gained massive popularity in the last year. To read more about Pulumi, check the official website at `https://www.pulumi.com/docs/get-started/`.

To orchestrate the whole infrastructure management process, CI/CD tools define the engines to automate everything across different cloud environments.

Now that we have covered the different pieces of the hybrid cloud puzzle, we will focus on how to connect the dots between two cloud environments – in our case, OpenStack and AWS.

Preparing for a hybrid model

Establishing a hybrid cloud architecture first requires an adequate network connection between the private and public clouds. Such a network connection is critical to maintaining business continuity. For that reason, continuous evaluation of the connection that looks at latency, uptime, and bandwidth is a must. Security is the other essential aspect of the hybrid connection. Depending on the hardware and network service used for such a connection, operators should ensure encryption in transit is enabled and redundancy is considered. Implementing a cloud hybrid connection can be greatly simplified when using cloud network-managed services. As shown in the next simplified network diagram, an OpenStack private cloud endpoint is connected to an AWS **Virtual Private Cloud** (**VPC**) through two different connections:

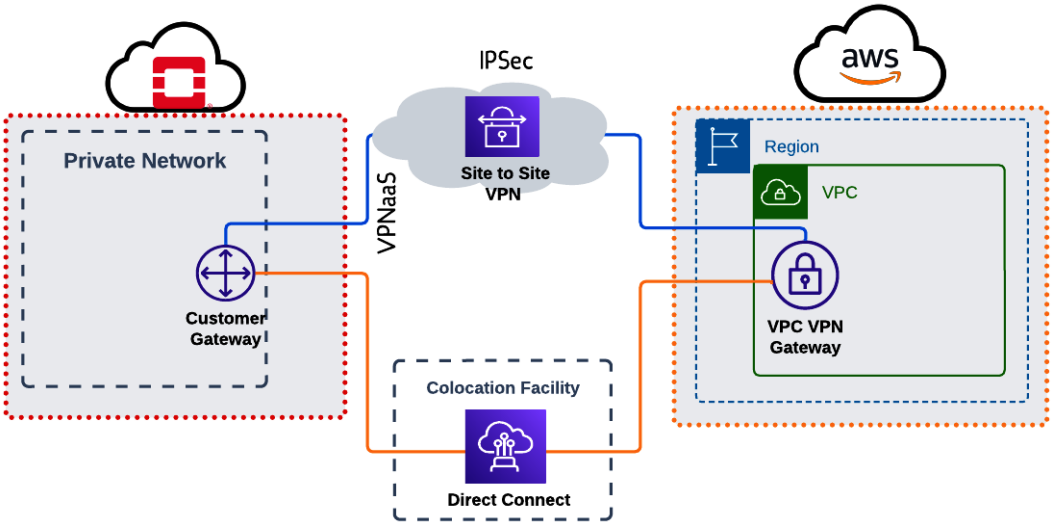


Figure 11.2 – A hybrid connection between the OpenStack and AWS clouds

The type of connection in the upper part of the figure illustrates a site-to-site VPN connection over the internet using an IPSec tunnel. Without exposing any resources in both cloud environments (the OpenStack private network range and AWS VPC) to the public world, communication occurs in an encrypted tunnel traveling over the internet. The implementation of a VPN connection between an OpenStack tenant environment and AWS VPC is much more automated than in a classic data center. As we covered in *Chapter 6*, *OpenStack Networking – Connectivity and Managed Service Options*, Neutron offers a managed network service, VPNaaS, that can be deployed as a site-to-site VPN connection in minutes. AWS offers a similar managed VPN feature as part of its VPC service that can establish a VPN connection by simply creating a VPN gateway and configuring a customer gateway (an OpenStack endpoint).

Important note

When using a site-to-site VPN connection in AWS, two tunnels will be created for redundancy. Each tunnel terminates in a different Availability Zone. If one tunnel is down, traffic is automatically shifted to the second active tunnel.

VPN can be the quickest way to establish such a hybrid cloud connection. A major pitfall of a site-to-site VPN connection linking two different locations across different regions is limited performance. In AWS' case, each VPN tunnel can support a bandwidth of up to 1.25 Gbps. Other public cloud providers might offer different options. For example, Azure allows cloud users to specify the type of VPN gateway they require, and depending on its size, the bandwidth can vary between 100 Mbps and 10 Gbps.

On the OpenStack side, Neutron will provision a virtual appliance with a certain bandwidth limit. Neutron offers a **Quality of Service** (**QoS**) feature that can be used to set network traffic policies for better network bandwidth control.

When dealing with complex and large-scale workloads that need to exist and move between different cloud environments, a limited bandwidth capacity can present a bottleneck. Moreover, a consistent connection between both cloud environments using a simple VPN connection cannot always be guaranteed. This is due to the long path taken across shared internet devices. For that reason, public cloud providers offer a second category network connection to establish a connection to a private environment. In the AWS world, **DirectConnect** is an ideal option when a stable, dedicated, isolated, and higher-performance connection is required.

Implementing a DirectConnect connection requires more steps in the deployment process, unlike a VPN connection setup. Cloud operators can order a DirectConnect connection via the AWS console or CLI and decide which type of connection they will put in place. DirectConnect offers two kinds of connections:

- **A hosted connection**: An AWS partner will handle the physical connection and link it to the device sitting at the edge of the OpenStack private cloud environment.
- **A dedicated connection**: This establishes a physical connection from the OpenStack endpoint to an AWS DirectConnect location. This type of connection comes with multiple bandwidth capacity options.

Which of these connectivity options you choose depends on many factors, such as cost, geographic location, and bandwidth throughput. To learn more about AWS DirectConnect technical details, check out the following AWS URL: `https://docs.aws.amazon.com/directconnect/latest/UserGuide/Welcome.html`

Hybrid connectivity requires a certain level of redundancy. If one link fails, you still have a second to carry the traffic, even with lower bandwidth throughput. The previous architecture diagram exposed a classic network design that leverages two connections, established using DirectConnect and VPN. Operators can configure their devices at the OpenStack private cloud end to primarily receive traffic

on the DirectConnect interface. That will terminate both connections into two different devices in the same location endpoint. If the DirectConnect connection fails for some reason, traffic will be switched to flow across the VPN connection as a backup. This design pattern is considered a good compromise that grants high availability at a fair price. If a hybrid connection requires the same consistent bandwidth with exact configurations in both links, two DirectConnect links will be put in place. That will maximize resiliency for critical production workloads. However, that comes of course with a higher bill. AWS recommends using dynamic routing to allow connections to fail over automatically and leverage the best available routes. This routing capability requires that the OpenStack hardware endpoint supports dynamic routing and auto-failover features.

Now that we have covered one of the most critical pieces of hybrid cloud design, networking, we will move on to the next part to cover different ways to design a containerized hybrid cloud setup.

Designing a containerized hybrid cloud

Container technology is the way to go to develop modern applications with all the benefits of portability, speed of deployment, and less operational overhead. The evolution of containerization has allowed organizations to run and scale their workloads in an almost zero-touch, reliable, and rapid way. With the rise of several forms of container engines such as Docker and LXD, a few container orchestration systems such as Kubernetes, Mesos, and Docker Swarm have come to light and empowered the way of developing applications. Today, we can find organizations deploying production workloads with more confidence than ever before. Container orchestration systems have shifted the gears on how applications are developed with the evolution of microservices architecture. Public cloud providers seized the opportunity to provide a platform to run a fully containerized environment. AWS, for example, offers (at the time of writing) three main container-related services, which are summarized as follows:

- **Elastic Container Service** (**ECS**): Deploys and manages containerized workloads through a fully managed container orchestration service
- **Elastic Kubernetes Service** (**EKS**): Runs and manages a Kubernetes control plane that is responsible for container deployment, scheduling, and availability
- **Fargate**: Acts as a serverless service that deploys and manages applications without the need to manage the underlying infrastructure

Azure has similar managed container services such as **Azure Container Instances** (**ACI**), **Azure Container Apps** (**ACA**), and **Azure Kubernetes Service** (**AKS**). Kubernetes has been a great deal for GCP, which has developed **Google Kubernetes Engine** (**GKE**) to reliably and efficiently deploy and scale containerized workloads on Kubernetes. The history of Kubernetes adoption could fill an entire book.

Deploying applications on containers in a hybrid cloud architecture requires a good understanding of a few design patterns. Organizations should seek the right tools that fit into their strategic initiative to take advantage of the hybrid model. In the following sections, we will explore a few implemented design patterns to deploy a hybrid containerized architecture using Kubernetes. Before tackling the different design models, we will pick up Kubernetes as a containerization platform. One of many

reasons to choose Kubernetes is that it is widely used, with plenty of features and tools available in its ecosystem. Another great benefit of adopting Kubernetes is its flexibility to design clusters and run containers, surrounded by several stable open source tools for container management across platforms, as well as its support federation feature that fits into a hybrid cloud architecture. Before delving into the previously mentioned design benefits of building a hybrid model, let's briefly look at Kubernetes jargon.

Kubernetes in a nutshell

Today, Kubernetes is everywhere. Eight years ago, companies were hesitant to start a production journey running Kubernetes, as it was new to the market since Google had recently made it publicly available. The concept of a container orchestration engine was not new at the time, since Docker Swarm and Mesos were quite evolved and widely used. Kubernetes, as expected, came with a different engine concept. Understanding the basic concepts of how it works is an essential step to confidently run it in production environments.

For container enthusiasts, using cloud services will facilitate the adoption of a containerization model based on Kubernetes. As mentioned in the previous section, cloud hyperscalers offer a managed service that supports Kubernetes. The Kubernetes PaaS model simply removes any overhead to manage and install the containerization orchestration engine. This also includes additional networking, storage, and availability aspects.

As shown in the following figure, Kubernetes is designed to run different types of nodes within a cluster:

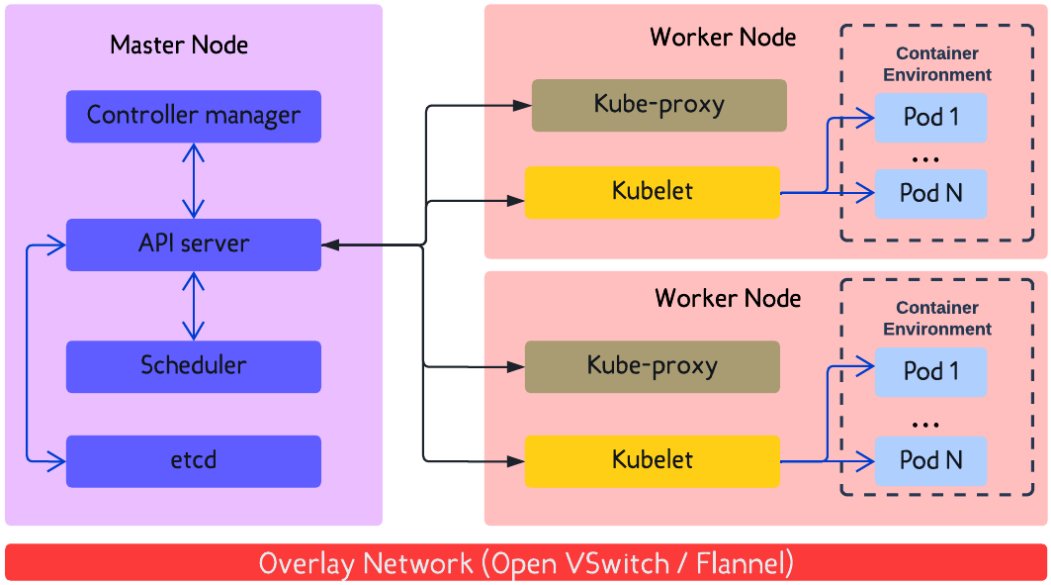


Figure 11.3 – The Kubernetes engine architecture at a high level

The node categories can be summarized as follows:

- **Master node**: This is responsible for running most of the essential processes that manage a Kubernetes cluster, including the following:
  - **Scheduler**: This runs the `kube-scheduler` process that manages the placement of containers on proper hosts based on resource availability and load.
  - **Controller manager**: This runs the `kube-controller-manager` process that keeps an eye on the cluster activities.
  - **API server**: This runs the `kube-apiserver` process that allows communication between a Kubernetes client and the cluster. The master node can be reached via the API server.
  - **etcd**: This is a key-value storage that handles the state of the cluster.
- **Worker node**: This runs the workloads in containers and is composed of two components:
  - **Kubelet service**: This runs the `Kubelet` process that allows cluster inter-communication. A Kubelet process also listens from the API server to manage containers in the worker node.
  - **Kube-proxy service**: This runs the `kube-proxy` service that handles networking between services in a Kubernetes cluster.

Kubernetes jargon introduces a few concepts as well as those related to its cluster-based architecture description, as illustrated in the following diagram:

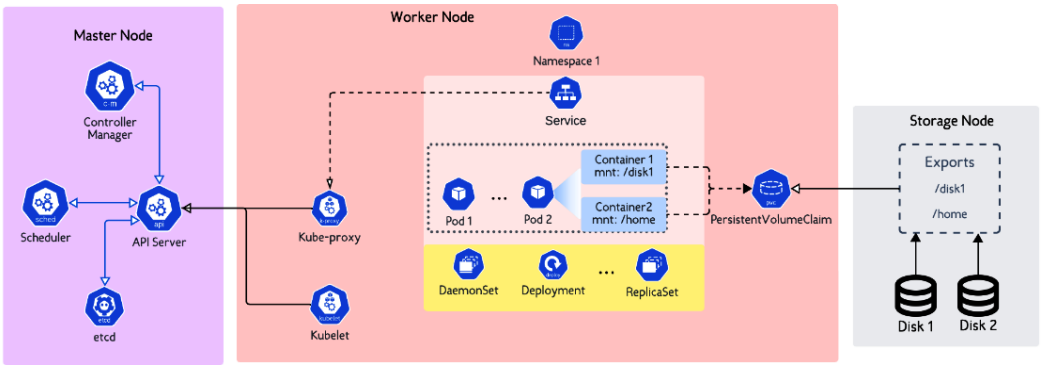


Figure 11.4 – The Kubernetes engine architecture at a low level

Different cluster components can be described as follows:

- **Pod**: This is defined as the smallest and most basic component in a Kubernetes environment. A pod combines and runs containers, attaches storage as required, and assigns a unique IP address for networking.

- **Controller**: This presents the main piece of the container orchestration engine. It manages all different aspects of a pod such as creation, deletion, replication, and rollouts. A Kubernetes controller keeps track of workload placements and allocated resources. There are different kinds of controllers, including **ReplicaSet**, **StatefulSet**, **DaemonSet**, **Deployment**, **Job**, and **CronJob**. Each controller type has a different use for specific pod management purposes. For example, the **ReplicaSet** controller creates a set of pods that all run the same application. **CronJob** is more dedicated to creating scheduled jobs. To read more about the Kubernetes controller types, refer to the official Kubernetes page at `https://kubernetes.io/docs/concepts/workloads/controllers/`.
- **Service**: This exposes an application that runs in one or multiple Pods for incoming requests. A Kubernetes service routes each request to the respective pod(s) and provides additional capabilities, such as DNS and load balancing to access a pod.
- **Namespace**: This is a scoped logical layer to isolate Kubernetes resource names, thus avoiding name collisions when many users launch several Kubernetes clusters in the *same* physical cluster.
- **Volume**: This provides persistent storage when required by a workload running in Pods. The latest version of Kubernetes supports more storage and volume features, as well as an exhaustive list of volume types. A full list of supported types of volumes can be found at `https://kubernetes.io/docs/concepts/storage/volumes/`.

Now that we have covered most of the important aspects of the Kubernetes world, we can verify how it fits into a hybrid cloud setup by exploring two of the most widely adopted deployment models.

Designing a decentralized models

The decentralized Kubernetes model, which is adapted for cloud hybrid architecture, is referred to as a *bursting* model. Organizations adopt such a design to overcome the over-provisioning of resources in their private clouds by leveraging public cloud resources. In such a layout, each cloud environment has its own Kubernetes infrastructure deployed and managed separately. In a second iteration, operators need to manage network connectivity between clusters that reside in both cloud environments. The next figure illustrates public cloud bursting in a hybrid setup where a Kubernetes cluster that runs in a private cloud extends its resources by reaching public resources (additional worker nodes):

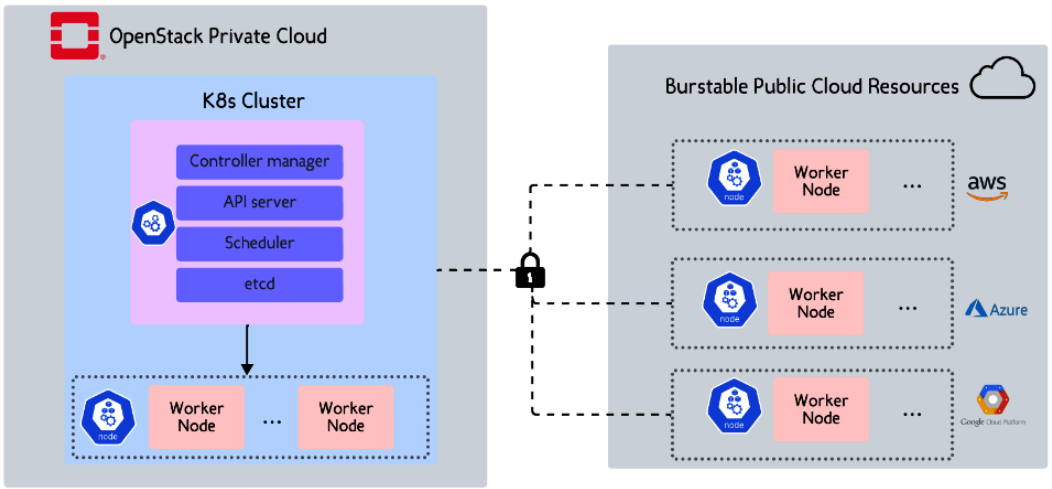


Figure 11.5 – A hybrid bursting model for OpenStack and a public cloud provider

Depending on each public Kubernetes managed service, operators will need to define extra parameters, such as DNS, cluster API endpoint, and namespace configurations, so that public resources can be used within the same workload or application context. The implementation of such a model can be far from complex and requires additional tweaks to properly synchronize the Kubernetes cluster resources, such as services and deployments. Operators must ensure that both clusters are consistent when it comes to configuration, identity management, and workload deployments.

As highlighted at the beginning of this section, AWS offers two managed services for workloads based on Kubernetes: EKS and Fargate. Before drafting a bursting Kubernetes layout joining the OpenStack and AWS environments, it is essential to recognize the use cases for each AWS EKS and AWS Fargate service, as follows:

- Use EKS if you need to do the following:
  - Provision a managed Kubernetes cluster by AWS without the need to manage its control plane and operate its underlying infrastructure and scaling and security configurations
  - Have more flexibility in the networking, storage, and scaling options
  - Deploy complex and large workloads
  - Have more control of its deployment of cross-regional, on-premises, and private and public clouds
  - Have a wider integration with AWS native services for advanced configuration, such as Elastic Load Balancing and AWS App Mesh

- Use Fargate if you *don't* need to do the following:
  - Provision a Kubernetes infrastructure and its associated resources
  - Manage workloads by provisioning resources (EC2 instances) but via task definitions instead
  - Run containers based on cluster and instance sizes but on the CPU and RAM resources you need instead
  - Not focus on managing the cluster nodes but the workloads instead
  - Include additional features that come with the Kubernetes engine

If you require advanced configurations that support a cloud-agnostic setup and more networking customization, then EKS would be a suitable choice for a hybrid layout.

Another key consideration is that when employing storage in a Kubernetes hybrid decentralized model, both storage (volume) endpoints should have the same data. There are several ways to ensure data synchronization across both storage endpoints by leveraging some public cloud-managed storage services. For example, a volume attached to Pods running in a Kubernetes cluster in OpenStack can be copied over a private connection to AWS using **DataSync** (`https://aws.amazon.com/datasync/`).

As shown in the following architecture diagram, DataSync requires an endpoint for the inbound data transfer from a private cloud storage endpoint. You can create an **Elastic File System** (**EFS**) as a destination location in your AWS VPC. The EFS location can be configured in the `DataSync` task so that any data being transferred from a volume located in the private cloud is synced in an EFS share. It is important to note that it is possible to mount an EFS filesystem on Fargate, but it is only limited to volume static provisioning, whereas, with EKS nodes, it is possible to use dynamic persistent volume provisioning.

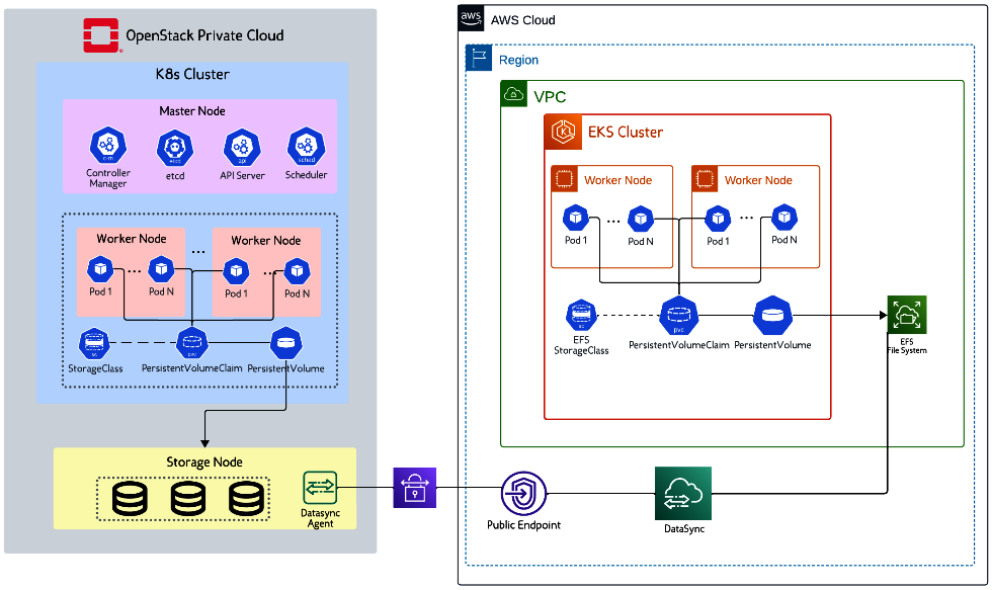


Figure 11.6 – A hybrid Kubernetes bursting model with volume synchronization

This hybrid model is suitable for workloads that do not require low latency with immediate scaling resources dynamically during peak demand. In such a model, cloud operators should expect more networking management, as well as ensuring data consistency across hybrid cloud environments. Managing multiple Kubernetes clusters across different cloud environments can be overwhelming, even when you employ public Kubernetes-managed services such as EKS and Fargate. The good news is that a few open source tools and wrappers based on the Kubernetes ecosystem have been developed to empower more extended designs. **Kubernetes Cloud Instance Provider** (**KIP**) enables you to manage Pods in cloud instances from one single and simplified interface. The next section will explore an abstracted layout that measures a decentralized hybrid Kubernetes layout between AWS and OpenStack using KIP.

Hybrid cloud bursting

Before tackling the idea behind KIP in this section, it is essential to cover briefly the term *Kubelet* as another key concept of the Kubernetes framework. The Kubelet is simply an agent that runs Kubernetes at the node level. It provides all kinds of features that deal with Pods, such as deployment, management, and communication between nodes and Pods. A Kubelet can be considered a coordinator between the Kubernetes control plane and the containers running on nodes.

> **Important note**
>
> To read more about the Kubelet, refer to the official Kubernetes page found at `https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/`

Based on the same Kubelet concept, the **Cloud Native Computing Foundation** (**CNCF**) has initiated an open source project that focuses on a **Virtual Kubelet**. The official Virtual Kubelet project web page can be found at `https://virtual-kubelet.io/`. It defines it as *"an open source Kubernetes kubelet implementation that masquerades as a kubelet."* The term *masquerades* is quite metaphoric, but one way to understand its concept is to consider the difference between a Kubelet and a Virtual Kubelet. The Virtual Kubelet provides more capabilities than the standard Kubelet by scheduling the provisioning of containers in other cloud solutions, but not on the nodes themselves. Under the hood, the Virtual Kubelet is highly customizable, allowing users to use other cloud environments and deploy Pods with their own APIs. The other piece of the Virtual Kubelet is the **Kubelet providers**. A Virtual Kubelet provider is a pluggable interface that provides all the necessary operational features and management of Pods and containers. A few of the providers commonly used to back Kubernetes nodes on cloud container platforms are AWS Fargate, Azure Container Instances, and OpenStack's Zun. An up-to-date list of supported providers can be found at `https://virtual-kubelet.io/docs/providers/#current-providers`.

The following diagram illustrates a simple cloud bursting example, from a private OpenStack cloud environment to the AWS public cloud. In an OpenStack tenant environment, a Kubernetes cluster runs alongside a Virtual Kubelet pod.

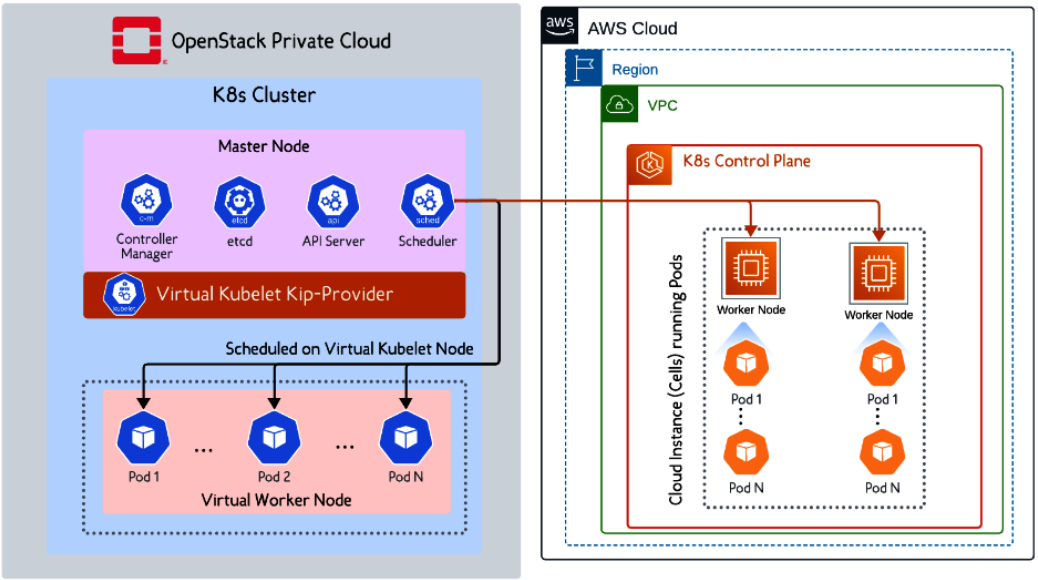


Figure 11.7 – Hybrid Kubernetes bursting between OpenStack and AWS via KIP

The other usage of a hybrid decentralized model is to deploy different Kubernetes clusters in more than one cloud environment (private and public) from a single deployer. This setup would require additional networking by connecting different clusters at the network layer. However, controlling and operating the clusters is performed individually. To centralize the management of workloads running across multiple cloud environments, we would need to provide a common control plane layer. This design brings the bursting model to the next level by adding a centralized control aspect to the hybrid model, which will be explored in more detail in the next subsection.

Designing a centralized model

The second hybrid model consolidates different Kubernetes clusters that run in both the private and public clouds. This type of architecture is referred to as **Kubernetes Federation**, which is primarily based on multi-cluster deployments with additional and common control planes. Under the Kubernetes Federation setup, one single source of a workload deployment is applied to a **host cluster** as a central deployment location that reaches different clusters across different connected cloud endpoints. The deployment of the application is then propagated to all environments with a view of a single target cluster. That removes any need to configure or maintain the state of an application in each cluster separately. The **Federation model** is a desirable hybrid option when you need to manage several clusters through a single API interface. An application is packaged and deployed from a single source. It spreads out its deployments equally across all visible clusters. From a host cluster (master configuration host) perspective, all connected clusters under its view are considered member clusters, so each deployed application will have its replicas throughout all worker nodes at scale. Kubernetes

Federation can also be customized during the application deployment. Some member clusters are part of different environments and might require specific local variables, such as network policies. Such member cluster-specific configurations can be supplemented by the host cluster holding the Federation configuration.

There are a variety of tools to deploy a hybrid Kubernetes centralized model. In the next section, we will focus on deploying a production-grade Kubernetes cluster across the OpenStack and AWS clouds using some trending open source solutions.

Deploying Kubernetes everywhere

There are several ways to orchestrate the deployment of Kubernetes clusters. Using **PaaS** and **SaaS** can be the fastest way that uses the least operational overhead. The open source world has also contributed to facilitating the deployment across clouds, but that involves preparation and understanding how each solution works under the hood. In the following section, we will focus on an open source solution named **Juju** from Canonical.

Juju integrates very well with a hybrid cloud deployment model and, more specifically, Kubernetes workloads across different cloud environments. Under the hood, a **Juju controller** host should exist per environment, whereas a **Juju client** performs deployment actions and other functions. As shown in the following diagram, a Juju cloud controller runs in each cloud environment and supports multiple cloud environments, including AWS and OpenStack.

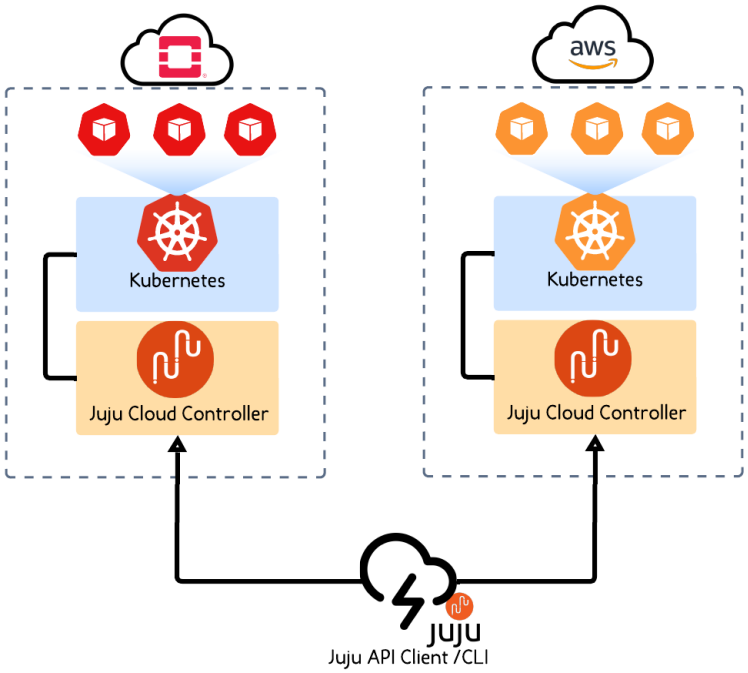


Figure 11.8 – A Juju workflow in a hybrid deployment model

The latest list of Juju's supported clouds can be found at `https://juju.is/docs/juju/juju-supported-clouds`. This architecture can be compared to the **Puppet** or **Chef** system management tools. Puppet uses master-slave architecture, in which the slave component can be compared to a Juju client and the master component is akin to a Juju cloud controller.

> **Important note**
>
> Several projects and open-source solutions exist that support hybrid and multi-cloud architecture using containers. Use cases of hybrid deployments using containers can be found at `https://openinfra.dev/hybrid-cloud/`.

Juju, similar to Puppet (manifests) or Chef (cookbooks), uses its own script operators, which are named **charms**. Charms deploy applications across different infrastructure resources including containers, virtual machines, and even bare metal machines. Juju charms have been designed to be flexible for deployment and can be written in several programming languages.

The following steps describe how a hybrid Kubernetes cluster can be deployed across public and private clouds:

1. Install the Juju client in a management/deployer host.
2. Configure the Juju client in the management host to bootstrap the AWS and OpenStack clouds.
3. Deploy the Juju cloud controller in the AWS public cloud.
4. Deploy the Juju cloud controller in the OpenStack private cloud.
5. Deploy Kubernetes clusters across both private and public clouds using Juju.
6. Connect both Kubernetes clusters.
7. Configure both Kubernetes clusters for federation.

In *step 1*, we will need to pick up a dedicated host for Juju cloud management and install the Juju client. This can be any machine that has connectivity to both the OpenStack and AWS environments. The Juju client supports several operating systems, a list of which can be found at `https://juju.is/docs/juju/install-juju`. The following step-by-step walkthrough installs the Juju client in a machine running on macOS:

```
$ brew install juju
```

To install the Juju client on a Linux machine that runs the Ubuntu operating system, use the following command lines:

```
$ sudo add-apt-repository ppa:juju/stable
$ sudo apt update
$ sudo apt install juju
```

The Juju client installation can be checked by running its interactive command line interface, as follows:

```
$ sudo juju
```

Here is the output:

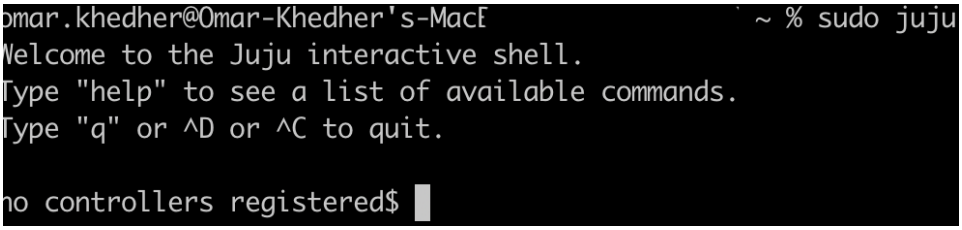


Figure 11.9 – Verifying the Juju installation

Important note

The Juju command line does not return any Juju cloud controller, as we have fresh-installed, and so far, no cloud controller in any cloud has been configured. Juju automatically registers each newly configured cloud controller, as we will see later.

You can also list the currently supported list of cloud environments by running the following command line:

```
$ sudo juju list-clouds --all
```

We get the following output:

```
You can bootstrap a new controller using one of these clouds...

Clouds available on the client:
Cloud                                                          Regions  Default           Type     Credentials  Source    Description
arn:aws:eks:eu-west-3:031300398971:cluster/eks-cluster-dev     0                          k8s      0            built-in  A local Kubernetes context
aws                                                            27       us-east-1         ec2      0            public    Amazon Web Services
aws-china                                                      2        cn-north-1        ec2      0            public    Amazon China
aws-gov                                                        2        us-gov-west-1     ec2      0            public    Amazon (USA Government)
azure                                                          44       centralus         azure    0            public    Microsoft Azure
azure-china                                                    4        chinaeast         azure    0            public    Microsoft Azure China
cloud_user@eks-acg.us-east-1.eksctl.io                         0                          k8s      0            built-in  A local Kubernetes context
equinix                                                        25       px                equinix  0            public
google                                                         25       us-east1          gce      0            public    Google Cloud Platform
localhost                                                      1        localhost         lxd      0            built-in  LXD Container Hypervisor
oracle                                                         50       af-johannesburg-1 oci      0            public    Oracle Compute Cloud Service
```

Figure 11.10 – The Juju supported cloud providers list

The next step is to bootstrap our cloud environments using Juju. To interact with AWS resources and bootstrap the first Juju cloud controller, we will need to generate a key pair of EC2 instances in AWS, constructed by access and secret keys.

> **Important note**
>
> OpenStack is not listed here, as Juju lists only public cloud providers. Private clouds, including OpenStack, would require adding it via the Juju command-line interface, which will be covered in the following sections.

The following part of the walkthrough requires you to have an AWS account and an **Identity Access Management** (**IAM**) user with programmatic access. Access your AWS account and make sure that you have sufficient IAM permissions to create an IAM user with programmatic access. From the main console services tab, click **IAM** and select **Users** from the **Access management** drop-down option list in the bottom-left corner of the console:

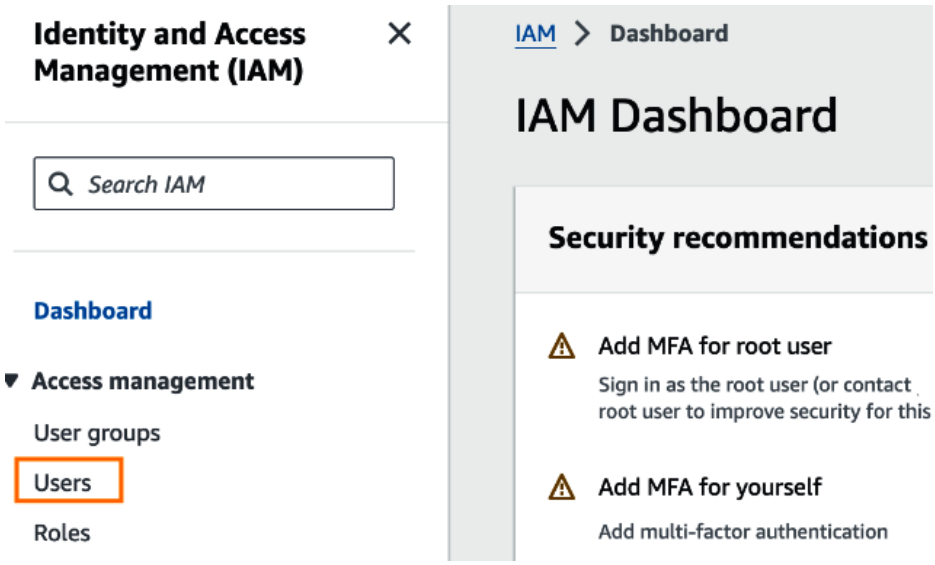


Figure 11.11 – The AWS IAM console

Next, click on the **Create user** button:

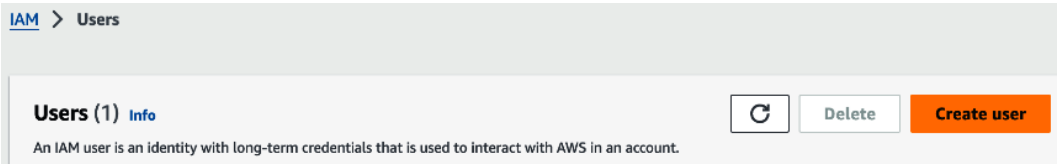


Figure 11.12 – Creating an IAM user in the AWS console

Provide an IAM user in the **User name** field and click on **Next**. Leave the **Provide user access to the AWS Management Console** checkbox unchecked, as we just need a user with programmatic access:

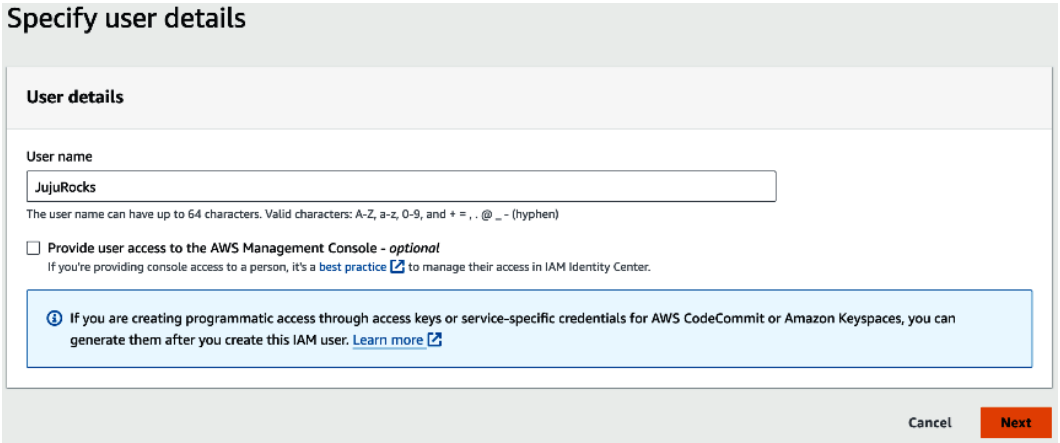


Figure 11.13 – Specifying user details in the AWS console

The next step involves allowing our Juju user to perform on AWS resources. The Juju client will bootstrap a cloud controller in an EC2 instance. For the sake of simplicity and based on the least privilege principle, we will assign an AWS-managed policy with full permissions on EC2 resources in AWS by attaching an **AmazonEC2FullAccess** policy to the created user, as follows:

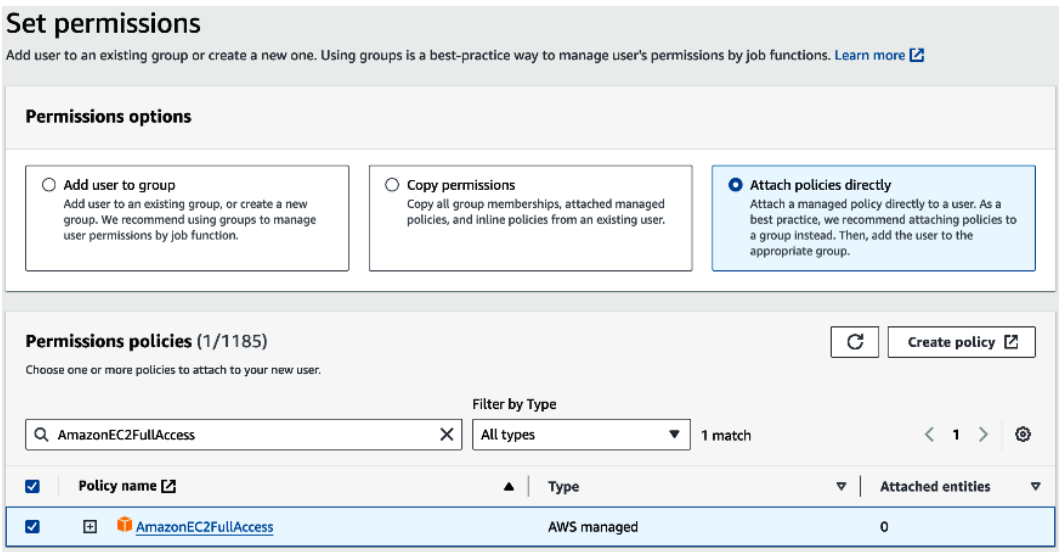


Figure 11.14 – Assigning an IAM policy to the Juju IAM user in the AWS console

Important note

From a best security practice perspective in AWS, it is not recommended to attach a full array of permissions to an IAM user or role. Custom policies can restrict the allowed actions on services and resources. As a rule of thumb, always use the least privilege principle when creating policies and assigning permissions.

Click on **Next** and on the **Review and create** page, click on **Create user**:

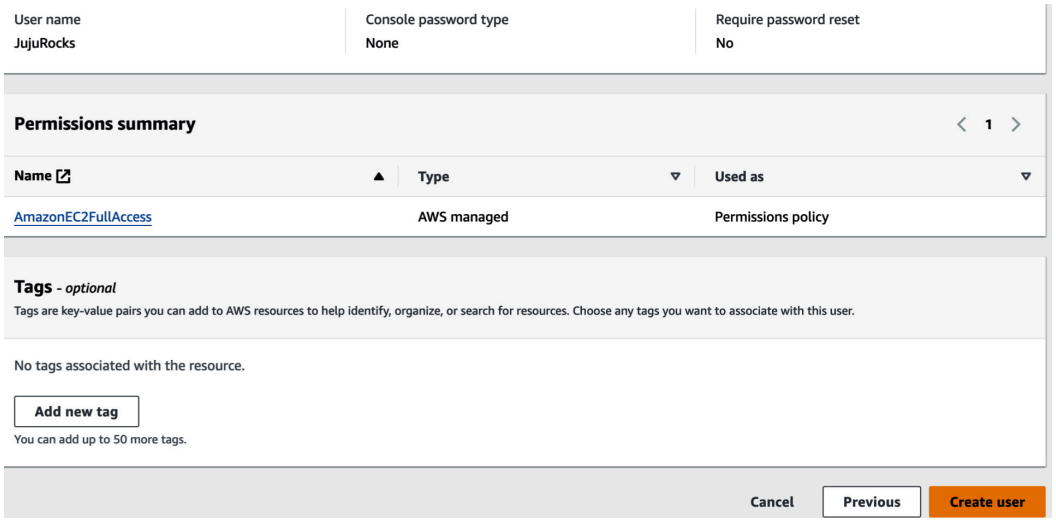


Figure 11.15 – Reviewing and creating the Juju IAM user in the AWS console

On the **IAM | Users** dashboard, select the newly created Juju IAM user. In the **Summary** tab, click on the **Create access key** link:

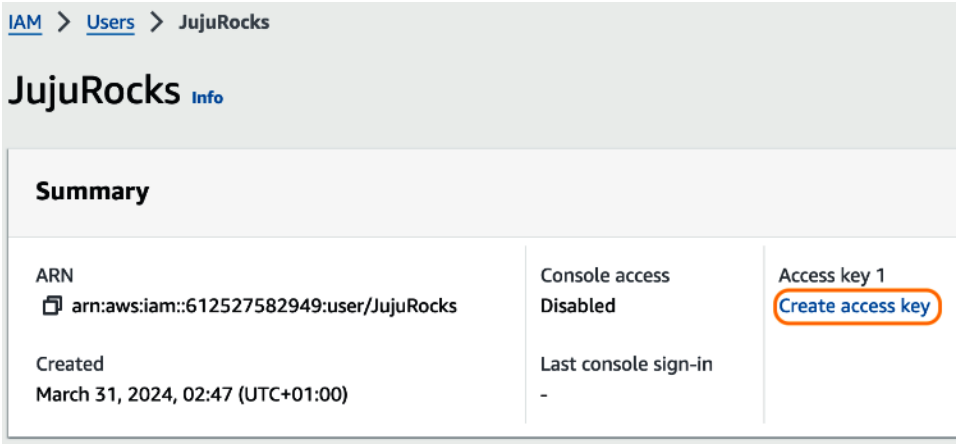


Figure 11.16 – Creating an access key for the Juju IAM user in the AWS console

The next page in the IAM user wizard will require you to select the credentials use case. Select the **Application running outside AWS** option:

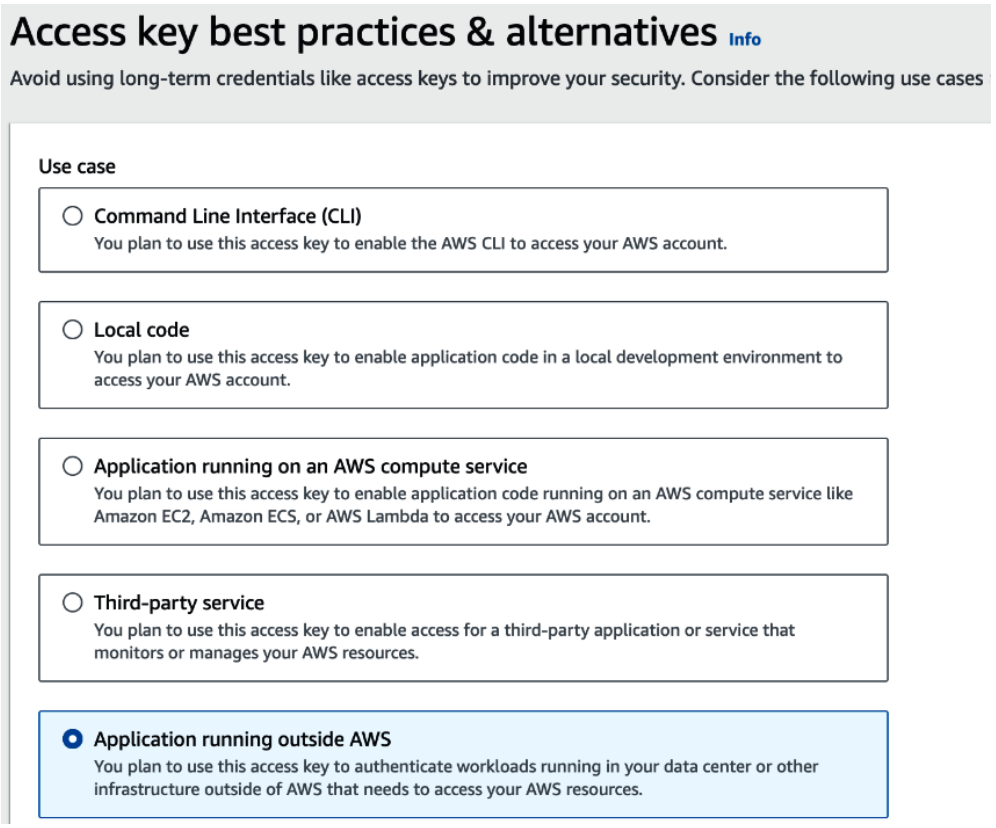


Figure 11.17 – Selecting an access key use case for the Juju IAM user in the AWS console

The last step of the creation of the IAM user credentials is optional: providing a tag value for the AWS credentials resources. Add a tag value and click on **Create access key**:

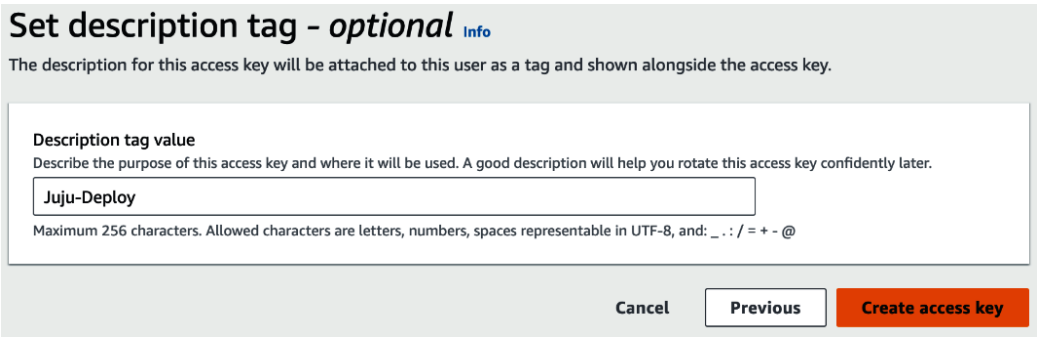


Figure 11.18 – Adding an AWS tag resource for the Juju IAM user in the AWS console

This will generate a pair of access and secret keys. You can copy and save them or download a CSV file containing the credentials in a secure location locally:

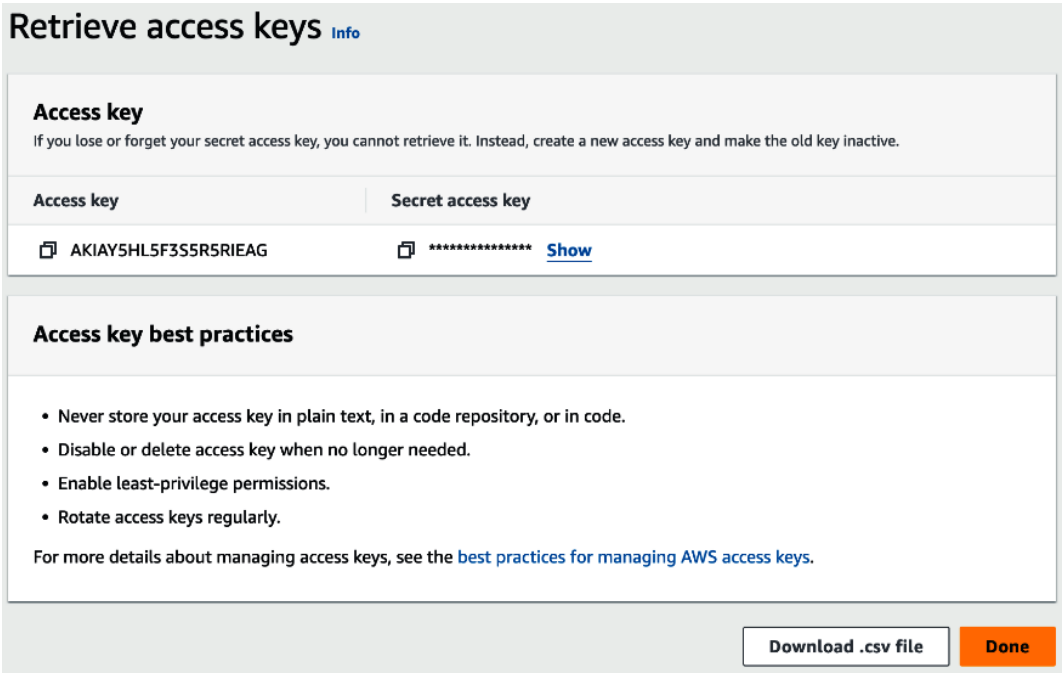


Figure 11.19 – Retrieving the generated keys for the Juju IAM user in the AWS console

Switch back to the Juju command-line interface in the management host and run the following command line to provide a key pair, including the access and secret keys created in the previous step:

```
$  sudo juju add-credential aws
```

The output is as follows:

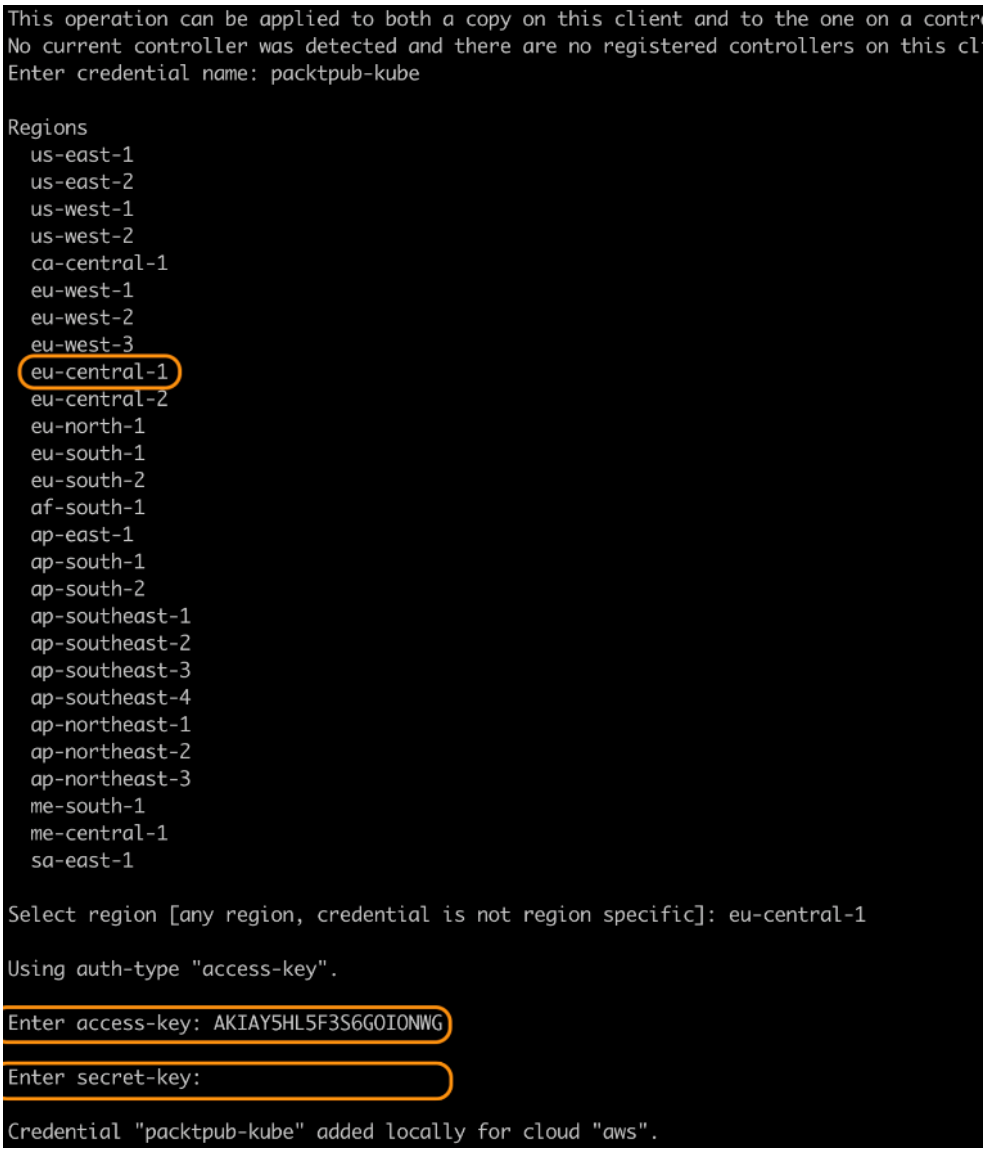


Figure 11.20 – Adding AWS IAM user cloud credentials using the Juju client

The previous command line will optionally request which region Juju will be operating within by providing the generated access and secrets previously. Juju will create a local AWS profile named `packetpub-kube`.

The next step is to create our first Juju cloud controller in AWS by running the following command line:

```
$ sudo juju  bootstrap aws juju-controller
```

We get the following output:

```
Creating Juju controller "juju-controller" on aws/us-east-1
Looking for packaged Juju agent version 3.4.0 for arm64
Located Juju agent version 3.4.0-ubuntu-arm64 at https://juju-dist-aws.s3.amazonaws.com
Launching controller instance(s) on aws/us-east-1...
 - i-0fe121e63055b51dc (arch=arm64 mem=4G cores=1)
Installing Juju agent on bootstrap instance
Waiting for address
Attempting to connect to 34.233.124.117:22
Attempting to connect to 172.31.20.181:22
Connected to 34.233.124.117
Running machine configuration script...
```

Figure 11.21 – Creating a Juju cloud controller in AWS

Under the hood, Juju performs an API call to the AWS environment and spawns an EC2 instance. The cloud controller instance can be checked in the AWS console in an initializing status from the EC2 dashboard:

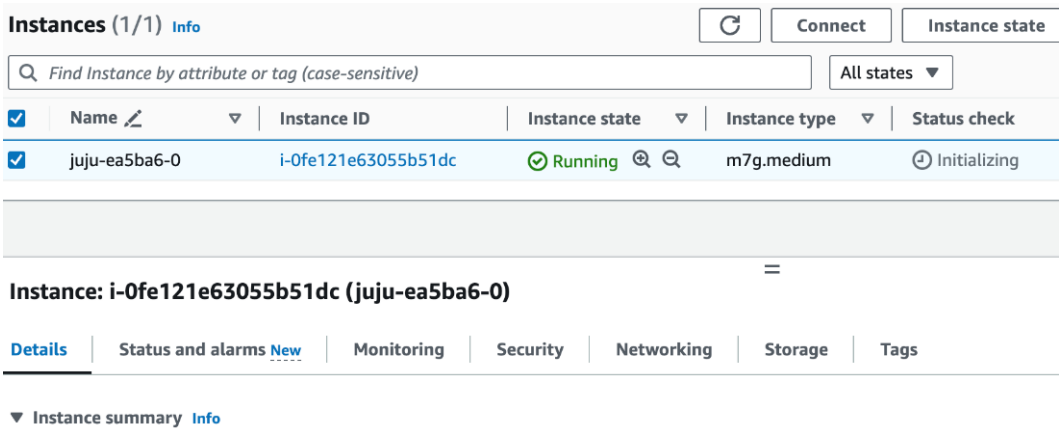


Figure 11.22 – A Juju cloud controller EC2 instance created in the AWS console

Note that Juju bootstraps the Juju controller with default configurations, such as the size of the instance (`m7g.medium`). Make sure, in the Juju command line output, that the bootstrapping process completes, as shown in the following figure:

```
Bootstrap agent now started
Contacting Juju controller at 34.233.124.117 to verify accessibility...

Bootstrap complete, controller "juju-controller" is now available
Controller machines are in the "controller" model
```

Figure 11.23 – The completion of Juju cloud controller bootstrapping in AWS

Our created controller host is assigned a public IP (`34.233.124.117`) that the Juju client communicates with.

> **Important note**
>
> The Juju bootstrap command can be customized further by specifying the size of the instance and the region where the cloud controller will be deployed. That can be achieved by using the **constraint** option in the Juju command-line interface, which uses a key-value format. A full list of Juju constraints can be found at `https://juju.is/docs/juju/constraint`.

The Juju cloud controller can be listed by running the following command line:

```
$ juju list-controllers
```

Here is the output:

```
Controller        Model  User   Access     Cloud/Region   Models  Nodes    HA  Version
juju-controller*  -      admin  superuser  aws/us-east-1       1      1  none  3.4.0
```

Figure 11.24 – Listing the deployed Juju controller using the Juju CLI

Another amazing benefit of Juju deployment is the support of a user graphical interface that comes with several features, such as the interactive creation of controller systems and the status of deployments. The user interface link can be accessed by default via the assigned public IP of the EC2 instance, which is generated by running the Juju dashboard command line. If you are running Juju version 3.0 or later, you will need to add the `juju-dashboard` charm to the deployed controller.

> **Important note**
>
> Prior to version 3.0.0, the Juju dashboard charm was deployed by default when installing the Juju controller, and access details were generated by running the `$ juju guis` command line. The previous command line was dropped from Juju 3.0.0 and later versions, and was replaced by the `$ juju dashboard` command-line option.

As we aim to expose Kubernetes resources, we will add the Kubernetes dashboard charm. Switch to the created controller by running the following command line:

```
$ juju switch juju-controller
```

The output is as follows:

```
juju-controller-kuber:admin/controller -> juju-controller (controller)
```

Figure 11.25 – Switching Juju controllers using the Juju CLI

Deploy the `juju-dashboard` charm for Kubernetes on the selected controller:

```
$ juju deploy juju-dashboard-k8s dashboard
```

Next, link the dashboard to the selected controller as follows:

```
$ juju integrate dashboard juju-controller
```

Finally, make the dashboard accessible by running the following command line:

```
$ juju expose dashboard
```

The Juju dashboard for Kubernetes should be deployed on the controller instance and accessed by pointing to the generated URL and login credentials provided in the previous output.

> **Important note**
> The output of the dashboard link and associated login credentials are no longer valid. Make sure to use your own generated link and credentials to access the deployed dashboard.

Once the Juju client and cloud controller are configured, we can start by deploying the first Kubernetes cluster in AWS. Juju provides different alternatives to deploy a Kubernetes cluster in AWS by simply deploying the respective charm. That can be a cluster deployed via a managed Kubernetes service, such as EKS, or a custom Kubernetes cluster running on an EC2 instance. Any of those aforementioned alternatives can be deployed simply via the Juju charms. In the following steps, we will deploy a self-managed Kubernetes cluster in AWS. Juju charms come with a production-grade Kubernetes cluster that can be deployed by simply running the following command line in the deployer host:

```
$ juju deploy charmed-kubernetes
```

The `charmed-kubernetes` Juju charm deploys several components in AWS, including the following:

- Kubernetes master node
- Kubernetes slave node
- Certificate authority
- etcd
- Kubernetes API load balancer
- Containerd engine
- Virtual networking service running Calico

A full list with respective links for each component can be found at the Juju Charmhub at `https://charmhub.io/charmed-kubernetes`. The previous command line will deploy the Kubernetes cluster on the created cloud controller. We can check each component's status and details by running the following command line in the Juju deployer host:

```
$ juju status
```

Here is the output:

```
App                   Version  Status       Scale  Charm                 Store       Rev  OS      Notes
easyrsa               3.0.1    active           1  easyrsa               jujucharms    3  ubuntu
etcd                  3.1.2    active           3  etcd                  jujucharms   14  ubuntu
calico                3.19     maintenance      4  calico                jujucharms    5  ubuntu
kubeapi-load-balancer 1.10.0   active           1  kubeapi-load-balancer jujucharms    3  ubuntu  exposed
kubernetes-master     1.6.1    active           1  kubernetes-master     jujucharms    6  ubuntu
kubernetes-worker     1.6.1    active           3  kubernetes-worker     jujucharms    8  ubuntu  exposed

Unit                     Workload     Agent  Machine  Public address  Ports           Message
easyrsa/0*               active       idle   0        34.233.124.117                  Certificate Authority connected.
```

Figure 11.26 – Listing the K8s cluster status in AWS using the Juju CLI

Alternatively, a new Kubernetes cluster can be checked through the installed dashboard. First, make sure that you have `kubectl` installed locally:

```
$ brew install kubectl
```

Once installed, run the `kubectl` proxy command line locally on `127.0.0.1` and port `8001`:

```
$ kubectl proxy --address='127.0.0.1' --port=8001
```

In a browser, enter `http://localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/`. This will show you a graphical user interface showing different deployed K8s cluster components as follows:

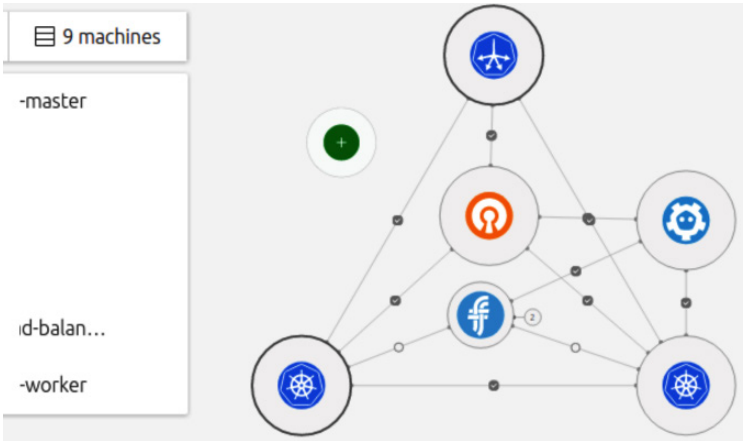


Figure 11.27 – The Juju dashboard for the K8s cluster deployed in AWS

Once we have achieved a fully deployed and running Kubernetes cluster in AWS, we can move to do the same steps in the OpenStack private cloud.

The `juju list-clouds` command line, as mentioned earlier, does not return OpenStack as a supported cloud by default. As a first step, add the OpenStack private cloud to Juju by running the following command line:

```
$ juju add-cloud
```

The output we get is this:

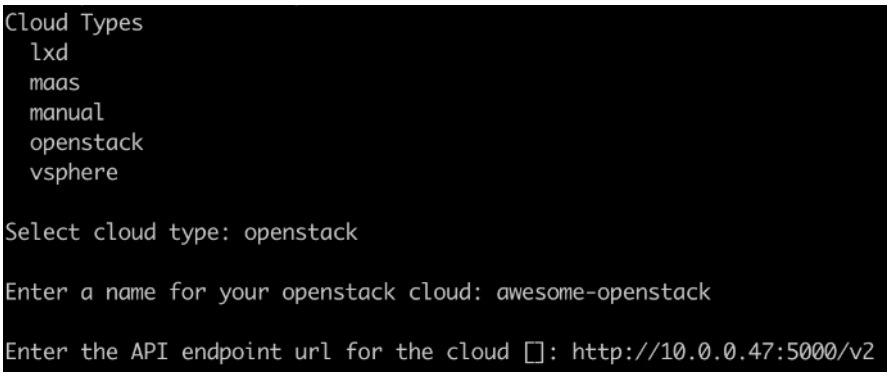


Figure 11.28 – Adding the OpenStack cloud using the Juju client

Once the OpenStack cloud is added, use the same `juju add-credentials` command line to authenticate against the OpenStack API:

```
$ juju add-credential awesome-openstack
```

Run the following command line to bootstrap the OpenStack cloud in Juju:

```
$ juju bootstrap awesome-openstack juju-controller2
```

As with the bootstrapping process in the AWS world, Juju will create a new cloud controller instance in OpenStack named `juju-controller2`.

Once created, we can deploy a Kubernetes cluster using the same Juju charm deployed in AWS. Use the `juju deploy` command line targeting the OpenStack Juju model as follows:

```
$ juju deploy charmed-kubernetes -m juju-controller2
```

The deployment process can take several minutes, and it can be checked by running the Juju status command line with the controller name as a flag, as follows:

```
$ juju status -m juju-controller2
```

This will be the output:

```
App                    Version  Status       Scale  Charm                  Store       Rev  OS      Notes
easyrsa                3.0.1    active           1  easyrsa                jujucharms    3  ubuntu
etcd                   3.1.2    active           3  etcd                   jujucharms   14  ubuntu
calico                 3.19     maintenance      3  calico                 jujucharms    5  ubuntu
kubeapi-load-balancer  1.10.0   active           1  kubeapi-load-balancer  jujucharms    3  ubuntu  exposed
kubernetes-master      1.6.1    active           1  kubernetes-master      jujucharms    6  ubuntu
kubernetes-worker      1.6.1    active         1/3  kubernetes-worker      jujucharms    8  ubuntu  exposed

Unit                      Workload     Agent       Machine  Public address  Ports           Message
easyrsa/0*                active       idle        0        10.0.0.22                       Certificate Authority connected.
etcd/0                    active       idle        1        10.0.0.54       2379/tcp        Healthy with 3 known peers.
etcd/1*                   active       idle        2        10.0.0.132      2379/tcp        Healthy with 3 known peers. (leader)
etcd/2                    active       idle        3        10.0.0.65       2379/tcp        Healthy with 3 known peers.
kubeapi-load-balancer/0   active       idle        4        10.0.0.88       443/tcp         Loadbalancer ready.
kubernetes-master/0       active       idle        6        10.0.0.55       6443/tcp        Kubernetes master services ready.
kubernetes-worker/0       active       idle        7        10.0.0.66                       Kubernetes worker running.
kubernetes-worker/1       waiting      allocating  8        10.0.0.43                       waiting for machine.
kubernetes-worker/2       waiting      allocating  9        10.0.0.103                      waiting for machine.

Machine  State    DNS           Inst id        Series  AZ  Message
0        started  10.0.0.22     juju-4762e1-0  xenial      Running
1        started  10.0.0.54     juju-4762e1-0  xenial      Running
2        started  10.0.0.132    juju-4762e1-0  xenial      Running
3        started  10.0.0.65     juju-4762e1-0  xenial      Running
4        started  10.0.0.88     juju-4762e1-0  xenial      Running
5        started  10.0.0.55     juju-4762e1-0  xenial      Running
6        started  10.0.0.66     juju-4762e1-0  xenial      Running
7        pending  10.0.0.43     juju-4762e1-0  xenial      Running
8        pending  10.0.0.103    juju-4762e1-0  xenial      Running
```

Figure 11.29 – Checking the K8s cluster status in OpenStack using the Juju client

As we deployed the same Kubernetes Juju charm in OpenStack as AWS, we should expect to see the same components listed in AWS up and running after the process has finished successfully. The previous command line provides the current state of each unit of the Kubernetes cluster. The cluster is fully deployed when all agents in each unit have the `idle` value.

> **Important note**
>
> The deployment process can take a while, and a real-time view of the deployment can be performed by running a `watch -c juju status` command line.

Once the cluster is deployed, it can be accessed from either the Kubernetes master or worker node. To take control of the cluster, we will need to set up the client and credentials access. Create a dedicated config directory in the deployer workstation:

```
$ mkdir -p ~/.kube
```

The next step aims to centralize the management of both clusters created in AWS and OpenStack through the same configuration location. To do this, we will copy the configuration files of each Kubernetes master deployed in both the AWS and OpenStack environments over an `scp` command line. The

following command lines will switch the Juju controller to the respective Kubernetes environment and copy the configuration file to the `~/.kube` directory. Let's start with the AWS Juju controller:

```
$ juju switch aws
$ juju scp kubernetes-master/0:config ~/.kube/aws-config
```

Run the previous command but switch to the OpenStack Juju controller:

```
$ juju switch awesome-openstack
$ juju scp kubernetes-master/0:config ~/.kube/os-config
```

To query the Kubernetes resources for both cloud environments, we can run the `kubectl` native tool that is, by default, installed in each master and worker node. In the management host, make sure that `kubectl` is installed (if this was not done previously) by running the following command line on a macOS machine:

```
$ brew install kubectl
```

> **Tip**
> The Kubectl setup guide for other platforms is detailed on the official Kubernetes website page at `https://kubernetes.io/docs/tasks/tools/#kubectl`.

We can then query each Kubernetes cluster from the Juju client machine by specifying the respective configuration path of each cloud environment. The following command line displays the status of the deployed Kubernetes cluster in AWS:

```
$ kubectl cluster-info --kubeconfig=~/.kube/aws-config
```

Here is the output:

```
Kubernetes master is running at https://34.233.124.117:443
Heapster is running at https://34.233.124.117:443/api/v1/namespaces/kube-system/services/heapster/proxy
KubeDNS is running at https://34.233.124.117:443/api/v1/namespaces/kube-system/services/kube-dns/proxy
Kubernetes-dashboard is running at https://34.233.124.117:443/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy
Grafana is running at https://34.233.124.117:443/api/v1/namespaces/kube-system/services/monitoring-grafana/proxy
InfluxDB is running at https://34.233.124.117:443/api/v1/namespaces/kube-system/services/monitoring-influxdb/proxy
```

Figure 11.30 – Listing the deployed K8s cluster components in AWS using kubectl

> **Important note**
> Alternatively, you can access the Kubernetes dashboard on the Juju client machine by running `$ kubectl proxy 127.0.0.1:8001`.

Now that we have two fully deployed Kubernetes clusters in AWS and OpenStack clouds, we will focus on connecting them. To use hybrid Kubernetes jargon, we will rely on the Federation feature.

Bringing clusters together

The Kubernetes Federation, a functionality that is known as **KubeFed**, enables workloads to run over multiple clusters, spread across multiple data centers and geographical locations. KubeFed leverages the control of different clusters from one central host that propagates any configuration changes to all or a selected member of the clusters. From an architecture perspective, the federated control plane leverages a set of standard APIs to control and manage cluster-wide operations. Since isolated clusters can be associated with DNS entries, KubeFed takes care of managing them by automatically configuring DNS for all discovered nodes. As shown in the next figure (in our case, with the hybrid model), KubeFed enables multiple clusters to be managed through a single control plane:

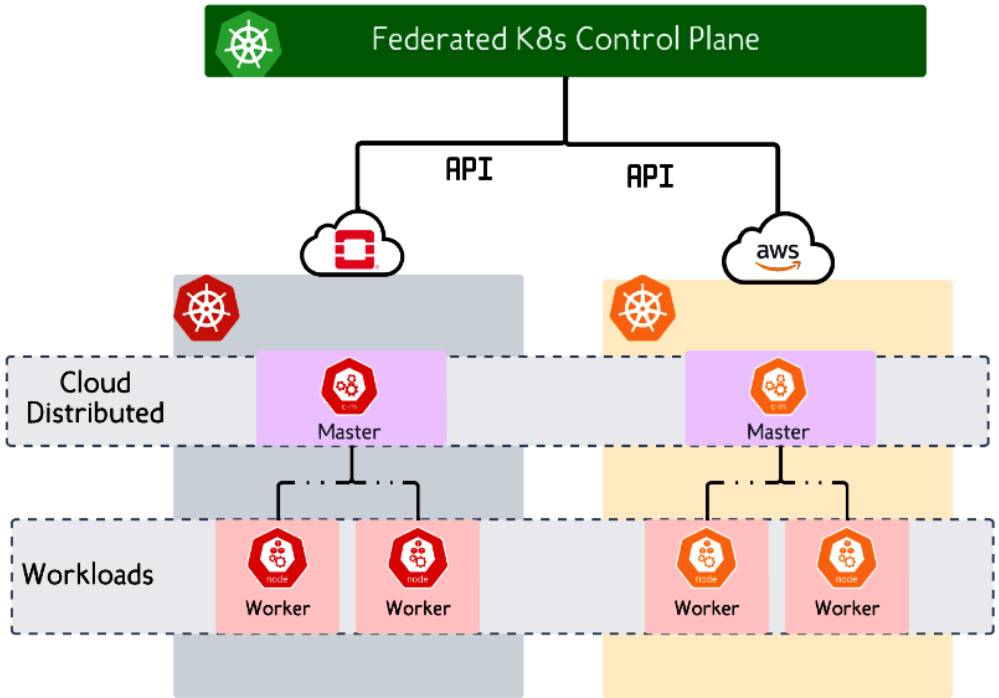


Figure 11.31 – A Kubernetes federation layout across OpenStack and AWS clouds

KubeFed can coordinate the state and configuration of many clusters running the standard Kubernetes API. In our use case, and for the AWS public cloud, any Kubernetes cluster deployed in EC2, EKS, or Fargate can be federated and linked to the cluster deployed in OpenStack's private cloud.

Before we look at the federation implementation, let's take a quick detour to look at some technical concepts of the KubeFed. The Kubernetes federation piece runs a regular Pod into a Kubernetes cluster on its own. **Federation Pods** are purely used for federation purposes and live within a created federated service. Functions such as service deployments, health monitoring, and DNS management are handled by the federated Pods.

The following steps will demonstrate how to use KubeFed to bring the AWS and OpenStack worlds together.

To start working with Federation, we will need to deploy a dedicated Kubernetes cluster on the host cluster. The simplest and quickest way to do so is by setting a Federation controller using Juju. Kubernetes' Federation cluster can be deployed in either OpenStack or AWS. You can also choose to deploy it in a different management environment and interact with it through a local client that can be installed later. For the sake of simplicity, we will use the OpenStack environment to run the Federation cluster. Using Juju, we can switch to the OpenStack private cloud environment and deploy a new federation cluster, using the same Kubernetes charm as earlier, as follows:

```
$ juju switch awesome-openstack
$ juju deploy charmed-kubernetes -m juju-controller-fed
```

Make sure to follow the status of the cluster deployment using the `$juju status` command line. Once the nine-component clusters are deployed, we can proceed to set up the federation cluster locally. With the same command lines we used for the OpenStack and AWS Kubernetes clusters, use `scp` to copy the cluster config file from any node under the ~/.kube/ directory:

```
$ juju scp kubernetes-master/0:config ~/.kube/fed-config
```

> **Tip**
> We copied the config file by renaming it, as we did with the AWS and OpenStack Kube files, to `fed-config`.

A good practice when dealing with multiple Kubernetes clusters is to centralize the management of all of them within a single command line. That will remove extra overhead when switching between clusters every time we need to operate a specific cluster. One way to do this is simply by merging all Kubernetes config files into one file. The latest releases of `kubectl` support the management of multiple cluster views. Start by creating an empty config file if it does not exist yet:

```
$ touch ~/.kube/config
```

Then, set the `KUBECONFIG` environment variable with a list of paths to all configuration files, as follows:

```
$ export KUBECONFIG=~/.kube/config:~/.kube/os-config:~/.kube/
aws-config: ~/.kube/fed-config
```

Once exported, run the following command line to merge all the `kubeconfig` files into one:

```
$ kubectl config view --flatten > ~/.kube/config
```

Using the `kubectl get-contexts` command line, we can display available contexts from the merged configuration step:

```
$ kubectl config get-contexts
```

Here is the output:

```
NAME                  CLUSTER
juju-controller-fed   awesome-openstack
juju-controller-2     awesome-openstack
juju-controller       aws
```

Figure 11.32 – A listing of the K8s contexts of the Juju controllers

> **Tip**
> We can display all clusters defined in the `kubeconfig` file by running the `kubectl config get-clusters` command line.

Before moving to construct the federated control plane, we will need to expose the federated service controller via DNS names. In our case, we have deployed the federation cluster in the OpenStack environment. Hence, a DNS provider configuration should be available to the federated service controller. When deploying the federated service controller in other cloud providers, DNS provider configuration is generated automatically if the deployed host cluster is the same as the DNS provider. For example, if the federated service controller is running in AWS and pre-configurated with a Route 53 hosted zone, the DNS provider will be automatically generated and pass DNS names to the federated service controller. In the following setup, we will configure **CoreDNS** as a federation DNS provider in the OpenStack federation cluster. To enable CoreDNS on `juju-controller-fed`, run the following command lines:

```
$ juju switch juju-controller-fed
$ juju config kubernetes-master enable-coredns=True
```

Then, create a `coredns-provider.conf` file that contains the IP address of the etcd service, obtained from the cluster status of the `juju-controller-fed` cluster and the DNS zone. The content of the `CoreDNS` file is configured as follows:

```
[Global]
etcd-endpoints = http://10.0.0.54:2379
zones = kube-fed.com.
```

> **Important note**
> The zone configured in this example is based on a local DNS configuration in the OpenStack environment. Feel free to use a custom DNS provider or a local one of your choice to create and manage zones for private cloud resources. Public cloud providers also offer a managed DNS service to create zones and DNS records, such as Route53 for AWS and Azure DNS for the Azure cloud.

Now, it is time to deploy the federation control plane on `juju-controller-fed`. We briefly discussed Kubefed, the tool for Kubernetes federation, but we have not installed it yet. To do so, run the following command lines on macOS to install Kubefed:

```
$ curl -LO https://github.com/kubernetes-sigs/kubefed/releases/
download/v0.10.0/kubefedctl-0.10.0-darwin-amd64.tgz
```

Next, extract binaries to one of the directories and set the necessary executable permissions for the Kubefed binary:

```
$ tar -xzvf kubefedctl-0.10.0-darwin-amd64.tgz
$ sudo cp kubernetes/client/bin/kubefed /usr/local/bin
$ sudo chmod +x /usr/local/bin/kubefed
```

Run the `kubefed init` command line to deploy and initialize the required service in the federation cluster:

```
$ kubefed init hybridfed --host-cluster-context=juju-controllerfed
--dns-provider="coredns" --dns-zone-name="kube-fed.com"
--dns-provider-config=coredns-provider.conf
```

Here is the output:

```
Federation API server is running at: 10.0.0.99
```

Figure 11.33 – Running the Kubefed Federation control plane

The `kubefed init` Federation command line will install a new API server by exposing the Federation service, as shown previously. It also takes care of installing the controller manager for Federation in addition to a dedicated namespace, referred to as `federation-system` in the host cluster. Now that we have Federation ready for usage, we can start controlling the Kubernetes clusters for Federation. Proceed by switching to the Federation context:

```
$ kubectl config use-context hybridfed
```

Next, add Kubernetes clusters running in OpenStack and AWS clouds to the joined Federation context. Let's start by adding AWS:

```
$ kubefed join aws --host-cluster-context=juju-controller-fed
```

The output is as follows:

```
cluster "aws" created
```

Figure 11.34 – Joining an AWS K8s cluster to the federation

The next command line will join the Kubernetes cluster running on OpenStack:

```
$ kubefed join awesome-openstack \
   --host-cluster-context=jujucontroller-fed
```

This is the output:

```
cluster "awesome-openstack" created
```

Figure 11.35 – Joining an OpenStack K8s cluster to the federation

Both previous command lines will add the necessary configuration in the Federation control plane so we can interact with the new construct through the Federating API as follows:

```
$ kubectl get clusters
```

The output is as follows:

```
NAME                STATUS AGE
aws                 Ready  3m
awesome-openstack Ready  3m
```

Figure 11.36 – Listing federated K8s clusters running in OpenStack and AWS

At this level, we have constructed a few federated Kubernetes clusters running in public AWS and private OpenStack clouds. Further operations on both clusters can be performed using the standard Kubernetes APIs. This federation enables the deployment of all the Kubernetes primitives such as namespaces, services, deployments, ConfigMaps, and so on. For example, through the same federating endpoint, we can deploy a Kubernetes namespace that will be spread across the hybrid cloud environment. Let's create a new YAML namespace called `fed-ns.yaml` with the following content:

```
apiVersion: v1
kind: Namespace
metadata:
  name: fed-ns
```

Then, run the following command line to create the namespace in the federating context as follows:

```
$ kubectl --context= hybridfed create –f fed-ns.yaml
```

Here is the output:

```
namespace "fed-ns" created
```

Figure 11.37 – Creating a federated K8s namespace

Let's look at the creation of the namespace in both the AWS and OpenStack worlds. Starting with AWS, simply switch the context by getting the list of deployed namespaces:

```
$ kubectl --context=aws get ns
```

The following is the output:

```
NAME              STATUS    AGE
ns/default        Active    1d
ns/kube-system    Active    1d
ns/fed-ns         Active    1m
```

Figure 11.38 – Listing the namespace for the K8s context running in AWS

Similarly, run the same command line but switch to the OpenStack context:

```
$ kubectl --context=awesome-openstack get ns
```

Here is the output:

```
NAME              STATUS    AGE
ns/default        Active    1d
ns/kube-system    Active    1d
ns/fed-ns         Active    55s
```

Figure 11.39 – Listing the namespace for the K8s context running in OpenStack

Congratulations! Our federation is ready to handle workloads in a hybrid setup across the OpenStack and AWS clouds. Deploying services and Pods will only require creating them via the federating context, and the federation controller will take care of sharing the services on both the private and public clouds.

Summary

This chapter explored more cloud opportunities with OpenStack. Leveraging container technology to deploy workloads across multiple cloud environments has been adopted by several organizations that seek the best usage of the *cloud* paradigm. This chapter highlighted some of the adopted approaches to deploying a hybrid cloud based on containers. The decentralized model can be considered a simple version of the centralized one where cloud operators can deploy workloads in multiple cloud environments. With the centralized model, firing one deployment from one place propagates a workload across multiple environments. The chapter demonstrated a use case of a hybrid model between AWS and OpenStack. Both worlds can meet, and having the capability to move between both cloud platforms is

a great *cloudy power*. Containerization allows companies to deploy, manage, and scale their workloads in more than a cloud environment, as we learned in this chapter. Federation mode has proven to be a rapid, reliable, and zero-touch way to operate workloads managed by Kubernetes, which we also learned about in this chapter. We further learned that OpenStack did not miss the chance again, and that with the rise of hybrid cloud adoption over the last few years, OpenStack has come to light. Due to the maturity of its ecosystem and rich set of APIs, OpenStack has become in demand to manage private cloud infrastructure. We learned that the marriage of both private and public clouds has, so far, been a successful experience. Let's keep it *cloudy*, where OpenStack keeps *shining*.

Index

Symbols

802.1Q tagging 176

A

active/active 210
active/passive 210
address scopes 195
Advanced Messaging Queuing Protocol (AMQP) 14
advertisement interval timer 224
alarming 91
AmazonEC2FullAccess policy 336
Amazon Web Services (AWS) 5, 302
amphorae 197
Anchore 72
reference link 73
Ansible 48-51
OpenStack environment management through 50
Anything as a Service (XaaS) 16
Aodh service 13, 91, 252
alerting with 255, 256
API endpoint configuration recommendations
reference link 316
application architecture patterns 319
Application as a Service (AaaS) 15
application development accelerator 16
application programming interface (API) 6
Artificial Intelligence (AI) 303
Authentication, Authorization, Accounting (AAA) 7
automatic failover 208
availability zones (AZs) 19
AWS Compliance
reference link 305
AWS Datasync
reference link 312
AWS Outposts 309
reference link 309
AWS Storage Gateway 312
Azure Container Apps (ACA) 324
Azure Container Instances (ACI) 324
Azure Deployment Manager 306
Azure Kubernetes Service (AKS) 137, 324
Azure Stack 309

B

Bare Metal as a Service (BMaaS) 15
Bedrock 303
BGP speaker 195

big data facilitator 18
block-level replication 215
Block Storage
defining 146, 147
block storage service 89, 90
Border Gateway Protocol (BGP) 195
Btrfs 151
bursting model 327

C

caching 93
capacity planning
building 33
Ceilometer 12
anatomy 252-254
ceilometer-api component 90
cells, infrastructure segregation
cellV1 125
cellV2 125-127
centralized logging 259
building, with OpenSearch 260, 261
centralized model
designing 331, 332
Kubernetes, deploying 332-347
Kubernetes Federation (KubeFed) 348-353
Ceph
deploying with 150-153
logical components, defining 151
Ceph cluster
installation link 151
certification-based replication (CBR) 216
charms 333
Chef 333
CI/CD tool 53
Cinder 11, 146
components 146
Cinder Driver Support Matrix
reference link 147
Cinder Scheduler Filters
reference link 154
Cinder Scheduler Weights
reference link 154
cloud
coding 47
containerizing 51-53
deploying 48
securing 46, 47
cloud application enabler 17
cloud architector, physical design 33
compute density 39, 40
CPU estimation 35
Memory estimation 36
Networking estimation 37, 38
parameters 34, 35
Storage estimation 36, 37
cloud architecture 26
conceptual design, drafting 27, 28
logical design, drafting 28, 29
cloud benchmarking
caching, deploying 276
Rally tool, using 276-280
cloud bursting 309, 312, 313
Cloud Deployment Manager 306, 321
CloudFormation 306, 321
Cloud Management Platform (CMP) 311
cloud management system (CMS) 184
Cloud Native Computing Foundation (CNCF) 330
cloud routing
configuring 188
Neutron dynamic routing 195, 196
tenant traffic, routing 188-195
cloud service provider (CSP) 33

cluster components
Controller 327
Namespace 327
Pod 326
Service 327
Volume 327
command-line interface (CLI) 7, 118
compute farm
expanding 142-144
compute segregation 120
infrastructure segregation 120
compute service 85
compute service components 116
API 117
compute 118
conductor 117
console access 118
scheduler 117
conceptual design
drafting 27, 28
constraint 342
Container Advisor (cAdvisor) 241
Container as a Service (CaaS) 16
container cluster management platform
Docker Swarm 136
Kubernetes 136
Mesos 136
containerization technology 319
containerized hybrid cloud
centralized model, designing 331, 332
decentralized models, designing 327, 329
designing 324
Kubernetes engine architecture 325
container orchestration engine (COE) 16, 116
Continuous Delivery (CD) 44
Continuous Deployment (CD) 44
Continuous Integration (CI) 44
Continuous Integration/Continuous Delivery (CI/CD) 16
CoreDNS 350
core plugin 174-176
Corosync 223
Create, Read, Update, and Delete (CRUD) 82
CronJob 327

D

DaemonSet 327
dashboard service 91, 92
database 92
caching 271
caching, deploying 272-275
empowering 270
Database as a Service (DBaaS) 13
DataSync
reference link 329
decentralized models
designing 327, 329
Hybrid cloud bursting 330
dedicated connection 323
dedicated instances feature 305
deployer
actioning 66-71
installing 63-66
deployment 327
extending 62
Deployment Manager 321
Destination Network Address Translation (DNAT) 37, 194
development environment
setting up 53
DevOps 44, 45

DevSecOps 45, 46, 316
cloud, securing 46, 47
OpenStack, deploying through 47
DirectConnect 316, 323
dedicated connection 323
hosted connection 323
Distributed Replicated Block Device (DRBD) 215
Distributed Virtual Routing (DVR) 185, 223
DNS as a Service (DNSaaS) 15
Docker 119
Docker Hub 142
Domain Name Service (DNS) 15
domain-specific language (DSL) 15
driver filter and weighing for scheduler
reference link 166
Dynamic Host Configuration Protocol (DHCP) 25

E

efficacy indicators 295, 296
Elastic Compute Cloud (EC2) 6
Elastic Container Service (ECS) 136, 324
Docker Swarm 136
Elastic Container Services Anywhere (ECS Anywhere) 312
reference link 312
Elastic Data Processing as a Service (EDPaaS) 15
Elastic File System (EFS) 329
Elastic Kubernetes Service Anywhere (EKS Anywhere) 312
reference link 312
Elastic Kubernetes Service (EKS) 136, 324
Elasticsearch, Logstash, and Kibana (ELK) 260
European Union (EU) 304
external network 32
Extract-Transform-Load (ETL) 18

F

Fargate 324
fault domains 121
fault tolerance (FT) 10
Federation model 331
Federation Pods 348
file share services
exploring 164
file-sharing service 90
Financial Operations (FinOps) 304
reference link 304
Flannel overlay network 138
flat 174
Fuxi 140

G

Galera 216
Galera MariaDB 270
General Data Protection Regulation (GDPR) 304
Generic Routing Encapsulation (GRE) 174
geo-redundancy 208
Git SCM plugins 65
Glance 9, 10
global efficacy 296
Gnocchi 252
data store 254, 255
Google Anthos 309
Google Cloud Platform (GCP) 5, 302
Google Kubernetes Engine (GKE) 137, 324
grandchild cells 126
graphical user interface (GUI) 13

H

HAProxy 94, 197
HAProxy load balancing
backend server 211
frontend server 211
HAProxy load balancing, algorithms
leastconn 212
round robin 212
source 212
uniform resource identifier (URI) 212
HAProxy load balancing, function layers
Layer 4 211
Layer 7 211
Havana 175
Heat 44, 321
Heat Orchestration Templates (HOTs) 15
high availability (HA) 12, 207
deploying 218-222
designing 210, 211
for database 213
for message bus 217, 218
HAProxy load balancing 211-213
measuring 208, 209
preparing 211
strategies 208
high availability (HA) for networking 222, 223
routing redundancy with DVR 227-230
routing redundancy with VRRP 223-226
high-performance computing (HPC) 6
Horizon 13
host cluster 331
hosted connection 323
HPC supporter 18
hybrid architecture
cloud-agnostic stack 320, 321
preparing 322-324
qualifying 320
hybrid cloud
enabling 311, 312
Hybrid cloud bursting 330
hybrid cloud computing
reference link 308
hybrid cloud environment 302
hybrid cloud models 309, 310
Hybrid Cloud services
reference link 312
hybrid cloud strategy 306-309
advantages 308, 309
hyperscaler 304
hypervisors 118, 119

I

ICMP access 192
Identity and Access Management (IAM) 317, 335
Identity Provider (IdP) 82, 120
identity service 81-84
image builder 53
image service 87
Index State Management plugin 262
Information Technology Infrastructure Library (ITIL) 33
Infrastructure as a Service (IaaS) 5, 301
Infrastructure as Code (IaC) 44, 302, 321
infrastructure automation 44
infrastructure monitoring 238
data, exploring 243-250
deploying 239-243
infrastructure segregation 120
availability zones 121, 122
cells 124

host aggregates 123, 124
regions 120, 121
infrastructure services
cloud software 238
under cloud 238
input/output operations per second (IOPS) 37
International Standards Organization (ISO) 304
internet service provider (ISP) 39
inventory 49

J

Job 327
Juju 332
Juju Charmhub
reference link 344
Juju client 332
Juju constraints
reference link 342
Juju controller 332

K

Keepalived 212
Kernel-based Virtual Machine (KVM) 29
Kernel Virtual Machine (KVM) 118
Keystone 7
Kolla-Ansible 52, 53
kolla-ansible infrastructure code 94
Kubectl
reference link 347
Kubelet
reference link 330
Kubelet providers 330
Kubernetes
deploying 332
Kubernetes Cloud Instance Provider (KIP) 330
Kubernetes engine architecture 325
Master node 326
Worker node 326
Kubernetes Federation 331
Kubernetes Federation (KubeFed) 348-353
Kuryr 140

L

L2 Population 176
labels 243
Liberty release 197
Libvirt API 240
Link Aggregation Control Protocol (LACP) 38
Linux Bridge 175
Linux Containers (LXC) 117
load balancer
as service 197-202
Load Balancer as a Service (LBaaS) 16, 197
local cloud
running 56-62
local environment
preparing 54, 55
logical design
drafting 28, 29
network segmentation approach 31
logical volume management (LVM) 146
deploying with 147-149

M

Magnum 95, 136-139
bay 137
BayModel 137
features 138

magnum client 138
pod 137
replication controller 138
service 137
managed services model 306
management network 32
Manila 11, 164, 166
core components 165
OpenStack core services, interacting 165
running 166-168
Manila Distributed and Shared File Systems solution 11
MariaDB database 240
Masakari 230
deploying 232-234
instance failover, managing 230-232
master/slave replication 213
maximum transmission unit (MTU) 38
mean time to failure (MTTF) 208
mean time to repair (MTTR) 48, 208
mechanism drivers 175
Memcached 271
message queue 92
Messaging as a Service (MSaaS) 15
microservices architecture 320
Mitaka 195
Modular Layer 2 (ML2) 174
mechanism drivers 176
type drivers 176
module libraries 49
modules 49
monitor daemon server (MON) 151
monitoring 90
Mouse, Keyboard, Mouse (MKM) 118
multi-chassis link aggregation (MLAG) 38
multi-cloud 301
multi-master replication manager (MMM) 214, 216
multiple backends
storage, expanding with 147
multiple writes concurrency pattern 270
Murano 115
MySQL multi-master replication with Galera 216
MySQL shared storage 214

N

National Aeronautics and Space Administration (NASA) 18
network address translation (NAT) 188
network bonding 38
Network File System (NFS) 147
deploying with 149, 150
network functions virtualization (NFV) 18
Networking as a Service (NaaS) 11
networking service 88
joining 197
load balancer as service 197-202
network types
external network 32
management network 32
storage network 32
tenant network 32
Neutron 11, 12
agents, demystifying 173
components, exploring 172, 173
features 12
network categories 174
Neutron dynamic routing 195, 196
Neutron plugins 174
core plugins 174-176
types 174
virtual switching, building 176

Neutron plugins, virtual switching
deploying, with OVN 185-188
OVN in OpenStack 182-184
OVS, deploying 179-181
traffic flow, with OVS 181, 182
vSwitch, opening in OpenStack 177-179
Non-OpenStack services 14
database, storing 14
other services 15, 16
Nova 8
Nova scheduler
filtering 117
weighting 117

O

Object Storage Devices (OSDs) 151
obkect storage service 90
Octavia 197
implementing, same load-balancing terms 197, 198
Octavia management network 199
OpenFlow 176
OpenSearch
deploying 261-267
used, for building centralized logging 260, 261
OpenSearch logging cluster
cluster manager node 261
coordinator node 261
data source node 261
ingest nodes 261
master-eligible node 261
OpenStack 5, 335
deploying 19-26
deploying, through DevSecOps 47
public cloud capabilities, using 310, 311
OpenStack-Ansible (OSA) 51
OpenStack, building blocks 6, 7
Aodh 13
Ceilometer 12
Cinder 11
Glance 9, 10
Horizon 13
Keystone 7
Manila 11
Neutron 11, 12
Nova 8
Placement 8, 9
Swift 10
OpenStack control plane 80
active/active 94
active/passive 94
alarming 91
block storage service 89, 90
compute service 85
dashboard service 91, 92
failure, preparing 93, 94
file-sharing service 90
identity service 81-84
image service 87
monitoring 90
networking service 88
obkect storage service 90
planning 95
running 80, 81
scheduling service 85-87
OpenStack-Omni 310
OpenStack public cloud marketplace
reference link 19
OpenStack services deployment 96, 97
advanced configuration 108, 109
cluster specs, listing 99
configuration 96
environment, parameterizing 107, 108
host groups, configuring 100-107

infrastructure code, preparing 99
network, initializing 97-99
staging environment, deploying 109-113
Open Virtual Network (OVN) 176
deploying with 185-188
Open vSwitch (OVS) 25, 175
in OpenStack 182-184
integration architecture 184
traffic flow with 181, 182
OSProfiler 287
implementing 287-289
OVS databases (OVSDBs) 183
OVS mechanism
deploying with 179-181

P

Pacemaker 212
Panko 252
pay-as-you-go model 304, 320
pets and cattle analogy 46
physical design
capacity planning, preparing 33
drafting 33
Placement 9
placement groups (PGs) 151
Platform as a Service (PaaS) 5, 301, 332
playbooks 49
Power Distribution Unit (PDU) 121
private cloud service provider 19
Prometheus 238, 239
Prometheus Libvirt exporter 240
provider network 32
public cloud service provider 19
public cloud strategy 302-306
Public Key Infrastructure (PKI) 83
Pulumi 306, 321
reference link 321
Puppet 333

Q

QEMU 118
Qpid 217
Quality Assurance (QA) 45
Quality of Service (QoS) 147, 185, 323

R

RabbitMQ 217
RADOS Block Device (RBD) 9, 150
Raft protocol 217
Rally tool
keystone under stress scenario 280-286
using, in cloud benchmarking 276-280
RDS Outposts
reference link 312
Redis 271
Relational Database Service (RDS) 312
Reliable Autonomic Distributed Object Store (RADOS) 150
Remote Desktop Protocol (RDP) 118
Remote Procedure Call (RPC) 23
repair point objective (RPO) 208
repair time objective (RTO) 208
ReplicaSet 327
Return On Investment (ROI) 304
Rexray 138
roles 49
routing redundancy with DVR 227-230
routing redundancy with VRRP 223-226

S

Sahara 115
Sarbanes-Oxley (SOX) 304
scheduling service 85-87
scoped token 81
Secret Manager as a Service (SMaaS) 15
security
  deploying with 72-77
security, in hybrid cloud 313, 314
  governance 314, 315
  network security 315, 316
  operation security, hardening 316, 317
service-level agreement (SLA) 50, 94, 208, 269
service plugin 174
shared drivers
  reference link 11
shared responsibility model 304
share drivers, Manila
  reference link 165
shared service 92
  caching 93
  database 92
  message queue 92
Simple Queue Service (SQS) 19
Simple Storage Service (S3) 9
single point of failure (SPOF) 10, 207
Software as a Service (SaaS) 5, 301, 332
software-defined networking (SDN) 12
software-defined storage (SDS) 145
software development life cycle (SDLC) 16
solid-state drive (SSD) 37, 86, 262, 270
Source Network Address Translation (SNAT) 37
split brain 120
SSH access 192
stacking approach 135
stateful service 210
StatefulSet 327
stateless service 210
storage
  expanding, with multiple backends 147
  filtering 154-157
  scheduling 154-157
storage network 32
subnet pools 195
Swift 10, 157, 160
  external API interface 160
  internal API interface 159
  logical core components 157
  replication interface 159
  running 160-164
  storage interface 159
Swift cluster
  building 158, 159
swift ring calculation
  reference link 161
system management tool 53

T

telemetry 251
  Aodh 252
  Gnocchi 252
  Panko 252
telemetry service
  deploying 257, 258
telemetry stack
  Aodh 255
  Ceilometer 252
  Gnocchi data store 254
tenant network 32, 174
tenant traffic
  routing 188-195
Terraform 306, 321

time to market (TTM) 17
toolchain
building, considerations 47
Top-of-Rack (ToR) 121
Total Cost of Ownership (TCO) 289
Transport Layer Security (TLS) 315
Trove 115
trunk-based software development 48

U

UML 119
undercloud monitoring tools 99
unscoped token 82

V

Vagrant Box
reference link 54
variables 49
version control system 53
Vertex AI 303
VirtualBox
download link 54
virtual IP (VIP) 94, 212
Virtual Kubelet 330
virtual local area network (VLAN) 38
Virtual Machine Monitors (VMMs) 118
Virtual Machine (VM) 269
virtual network identifier (VNI) 176
Virtual Private Cloud (VPC) 322
virtual private network (VPN) 12, 173, 316
Virtual Router Redundancy Protocol (VRRP) 212, 223
virtual switching
building 176
Virtuozzo 118
VM Workload Consolidation Strategy 291
vSwitch, in OpenStack
opening 177-179
VXLAN 174

W

Watcher 289
implementing 290-298
Watcher workflow
Monitor state 290
Optimizer component 290
Planner component 290
Web Application Firewall (WAF) 315
workload
monitoring 250, 251
workload segregation 127
filtering 131, 132
pre-filtering 128
resource provider allocation 128-131
weighting 132-136
write-through caching mechanism 271
WSGI module 280

X

XCP 119
Xen 119
XFS 151

Z

Zabbix agent 106
ZeroMQ 217
zero-trust 313
Zun 95, 139-141



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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

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

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

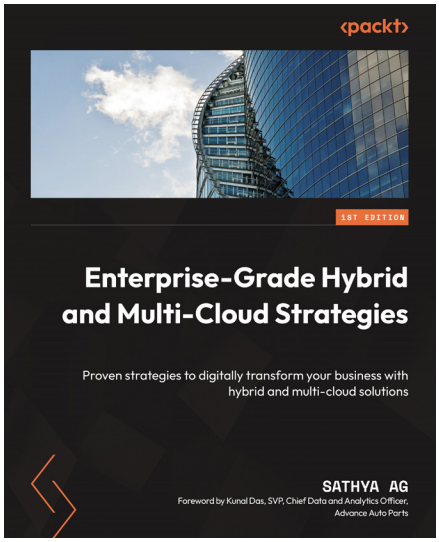


The Self-Taught Cloud Computing Engineer

Dr. Logan Song

ISBN: 978-1-80512-370-5

- Develop the core skills needed to work with cloud computing platforms such as AWS, Azure, and GCP
- Gain proficiency in compute, storage, and networking services across multi-cloud and hybrid-cloud environments
- Integrate cloud databases, big data, and machine learning services in multi-cloud environments
- Design and develop data pipelines, encompassing data ingestion, storage, processing, and visualization in the clouds
- Implement machine learning pipelines in a multi-cloud environment
- Secure cloud infrastructure ecosystems with advanced cloud security services



Enterprise-Grade Hybrid and Multi-Cloud Strategies

Sathya AG

ISBN: 978-1-80461-511-9

- Understand the hybrid cloud and multi-cloud paradigms
- Cultivate leadership will and mindset for crafting successful cloud transformation
- Design and architect a scalable and open foundation for a cloud-first IT organization
- Apply open standards and frameworks to design a vendor-neutral cloud foundation
- Understand the cloud adoption frameworks and conduct maturity assessments
- Realize tangible business value through cloud adoption initiatives

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Mastering OpenStack*, we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here to go straight to the Amazon review page` for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



`https://packt.link/free-ebook/9781835468913`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly