Towards Network Function Virtualization Orchestration

Thesis submitted in partial fulfillment of requirements
For the degree of
Master of Computer Application
of
Computer Science and Engineering Department
of
Jadavpur University

by

Kazi Anas Al Ahmed
Regn. No. - 154234 of 2020-2021
Exam Roll No. - MCA2360037

under the supervision of

Dr. Mridul Sankar Barik
Assistant Professor

Department of Computer Science and Engineering
Jadavpur University
Kolkata, West Bengal, India
2023

Certificate from the Supervisor

This is to certify that the work embodied in this thesis entitled **"Towards Network Function Virtualization Orchestration"** has been satisfactorily completed by **Kazi Anas Al Ahmed** (Registration Number 154234 of 2020 – 2021; Class Roll No. 002010503026; Examination Roll No. MCA2360037). It is a bona-fide piece of work carried out under my supervision and guidance at Jadavpur University, Kolkata for partial fulfilment of the requirements for the awarding of the **Master of Computer Application** degree of the Department of Computer Science and Engineering, Faculty of Engineering and Technology, Jadavpur University, during the academic year 2022 – 23.

Dr. Mridul Sankar Barik
Assistant Professor,
Department of Computer Science and Engineering,
Jadavpur University.
(Supervisor)

Forwarded By:

Prof. Nandini Mukherjee
Head,
Department of Computer Science and Engineering,
Jadavpur University.

Prof. Ardhendu Ghoshal
Dean,
Faculty of Engineering & Technology,
Jadavpur University.

Department of Computer Science and Engineering
Faculty of Engineering And Technology
Jadavpur University, Kolkata - 700 032

Certificate of Approval

This is to certify that the thesis entitled **"Towards Network Function Virtualization Orchestration"** is a bona-fide record of work carried out by **Kazi Anas Al Ahmed** (Registration Number 154234 of 2020 − 2021; Class Roll No. 002010503026; Examination Roll No. $MCA2360037$) in partial fulfilment of the requirements for the award of the degree of **Master of Computer Application** in the **Department of Computer Science and Engineering, Jadavpur University**, during the period of January 2023 to May 2023. It is understood that by this approval, the undersigned do not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein but approve the thesis only for the purpose of which it has been submitted.

Examiners:

(Signature of The Examiner)

(Signature of The Supervisor)

Department of Computer Science and Engineering
Faculty of Engineering And Technology
Jadavpur University, Kolkata - 700 032

Declaration of Originality and Compliance of Academic Ethics

I hereby declare that the thesis entitled **"Towards Network Function Virtualization Orchestration"** contains literature survey and original research work by the undersigned candidate, as a part of his degree of **Master of Computer Application** in the **Department of Computer Science and Engineering, Jadavpur University**. All information have been obtained and presented in accordance with academic rules and ethical conduct.

I also declare that, as required by these rules and conduct, I have fully cited and referenced all materials and results that are not original to this work.

Name: Kazi Anas Al Ahmed

Examination Roll No.: MCA2360037

Registration No.: 154234 of $2020 - 2021$

Thesis Title: Towards Network Function Virtualization Orchestration

Signature of the Candidate:

ACKNOWLEDGEMENT

I am pleased to express my gratitude and regards towards my Project Guide **Dr. Mridul Sankar Barik**, Assistant Professor, Department of Computer Science and Engineering, Jadavpur University, without whose valuable guidance, inspiration and attention towards me, pursuing my project would have been impossible.

Last but not the least, I express my regards towards my friends and family for bearing with me and for being a source of constant motivation during the entire term of the work.

Kazi Anas Al Ahmed
MCA Final Year
Exam Roll No. - MCA2360037
Regn. No. - 154234 of 2020 – 2021
Department of Computer Science and Engineering,
Jadavpur University.

Contents

List of Figures

Abstract

Network Function Virtualization (NFV) has emerged as a promising approach to transform traditional network infrastructures into more flexible, scalable, and cost-effective environments. By decoupling network functions from dedicated hardware appliances and virtualizing them in a software-centric manner. NFV Orchestration plays a critical role in realizing the benefits of NFV by automating the deployment, scaling, and lifecycle management of virtualized network functions (VNFs). In this project, we aim to explore the practical implementation of NFV Orchestration using OpenStack. By studying NFV, NFV Orchestration, and utilizing tools like OpenStack and Tacker, this project desires to contribute to the ongoing efforts in advancing the state-of-the-art in network virtualization and orchestration.

Chapter 1

Introduction

1.1 Background

Network Function Virtualization (NFV) promises an efficient way of managing and orchestrating Virtual Network Functions (VNFs), where VNFs can be dynamically instantiated or migrated based on current service requirements. This allows for more efficient resource utilization and the ability to adapt the network infrastructure to changing demands. For instance, in order to improve the performance of mission critical services, VNFs can be instantiated closer to users or even be migrated over time to follow mobile users.

1.2 Research Statement

Our research focuses on NFV orchestration using OpenStack, aiming to simplify the deployment and management of virtualized network functions (VNFs) for enhanced network flexibility and cost-efficiency. By using OpenStack's cloud computing platform, we will investigate its architectural capabilities and integration with NFV frameworks, ensuring different systems can work together smoothly. This research contributes to advancing network virtualization and orchestration, empowering network operators and service providers to leverage NFV's benefits using OpenStack.

1.3 Contribution of the Thesis

The thesis explores the various features of OpenStack. We study its different component and their uses. The thesis goes deeper into each part of OpenStack, like the compute, storage, and networking components, to understand what they can do. Furthermore, the thesis examines how OpenStack can be utilized for research purposes in the context of orchestration and cloud.

1.4 Outline of the Thesis

This thesis organised into seven chapters.

Chapter 1: Introduction

In Chapter 1, the thesis begins with an introduction to provide background information on the topic of network function virtualization (NFV). The research statement is presented to outline the main focus of the thesis, followed by a discussion on the contribution of the thesis. The chapter concludes with an outline of the structure and organization of the thesis.

Chapter 2: Network Function Virtualization

Chapter 2 starts with an introduction to NFV, defining the concept and exploring its relationship with software-defined networking (SDN). The benefits of NFV are discussed, highlighting its advantages in terms of flexibility, scalability, and cost-effectiveness. The chapter further explores the enablers and challenges of NFV and presents an architectural framework for implementing NFV. Additionally, several example use cases of NFV are provided, including network virtualization, mobile edge computing, orchestration engines, video analytics, security, and network slicing.

Chapter 3: NFV Orchestration

Chapter 3 introduces NFV orchestration, explaining the concept and its significance. The chapter provides an overview of NFV orchestrators, discussing their role and importance in managing virtualized network functions (VNFs). The VNF orchestration framework is explored, highlighting its components and functionalities. Furthermore, the chapter delves into the requirements of NFV orchestration, identifying the key factors for effective orchestration in NFV environments.

Chapter 4: Tools

Chapter 4 focuses on the tools used in NFV orchestration. It presents various open-source NFV orchestration platforms, including Open Source MANO (OSM), OPEN-O, CORD, and Cloudify. Additionally, the chapter discusses specific tools such as Mininet, Mini-NFV, OpenStack, Tacker, and TOSCA, which are commonly used in NFV deployments.

Chapter 5: OpenStack

Chapter 5 provides an overview of OpenStack, a widely adopted open-source cloud computing platform. The chapter explores the architecture of OpenStack, highlighting its core components and the logical relationships between them. It further examines the management aspects of OpenStack, including its operation interface, certification management, image management, computing management, storage management, network management, and orchestration management.

Chapter 6: Experiments

Chapter 6 focuses on practical experiments related to OpenStack. It covers the installation process of OpenStack and discusses the usage of OpenStack cloud services through graphical user interface (GUI), command line, and software development kit (SDK). The chapter also introduces the Python OpenStack SDK and discusses its architecture.

Chapter 7: Conclusion and Future Work

Chapter 7 concludes the thesis by summarizing the main findings and contributions. It reflects on the implications of the research and offers suggestions for future work and further exploration in the field of NFV orchestration and OpenStack.

Chapter 2

Network Function Virtualisation

2.1 Introduction

Nowadays, network operators have extended their network to a large number of areas. To accomplish this, they have installed a significant number of proprietary hardware appliances. Those hardware are many varieties, like server,router,switch etc. Now if the operator want to introduce a new network service then they may need to add new hardware appliances to their existing infrastructure.

These appliances could be specialized servers, routers, switches, or other networking equipment that are specifically designed to support the new service. However, as more and more of these appliances are added to the network, finding the physical space to house them becomes increasingly difficult. Additionally, the equipment requires a certain amount of power to operate, which adds to the overall power consumption of the network. Moreover, this challenge is further compounded by other factors, such as the increasing costs of energy, which means powering the additional hardware appliances can be expensive and adding new hardware to the network often requires a significant capital investment, which can be a challenge for network operators with limited budgets. Furthermore, designing, integrating, and operating increasingly complex hardware-based appliances requires a certain set of skills. Now it is very difficult to find such people with certain set of skill as hardware-based appliances become more specialized and complex. As a result, network operators may face difficulty to finding the right person to maintain the new hardware appliances.

Now hardware-based appliances life cycle ends quickly, which requires network operator to replace the entire appliances. For this the revenue is little and some time there is no revenue.

2.1.1 Definition

Network Function Virtualisation is refer to use virtual machines instead of hardware appliances. It is an approach to network architecture that aims to transform how network operators design and deploy their networks. It achieves this by consolidating many different types of network equipment onto industry-standard high volume servers, switches, and storage devices. In an NFV architecture, these functions are implemented as software applications that can be run on commodity hardware such as standard servers, storage devices, and switches. By consolidating network functions onto commodity hardware, network operators can reduce the cost and complexity of managing their networks. These virtualized network functions can be deployed in various locations, including data centers, network nodes, and even end-user premises. Overall, NFV allows network operators to leverage standard IT virtualization technology to build more flexible, scalable, and cost-effective networks that can adapt to changing network demands. It also enables operators to introduce new services more quickly and at a lower cost than with traditional hardware-based approaches. So we can that implementation of network functions in software is a core concept of Network Function Virtualisation (NFV).

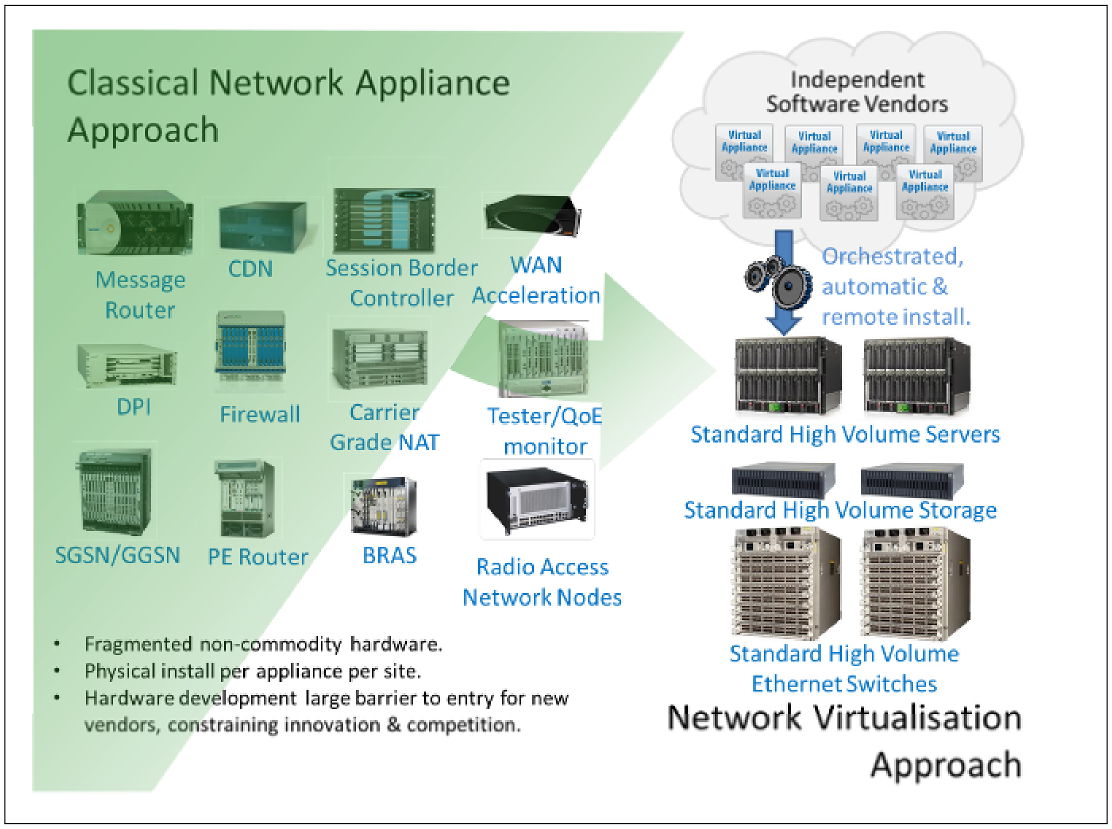

Classical Network Appliance Approach
Message Router
CDN
Session Border Controller
WAN Acceleration
DPI
Firewall
Carrier Grade NAT
Tester/QoE monitor
SGSN/GGSN
PE Router
BRAS
Radio Access Network Nodes
Fragmented non-commodity hardware.
Physical install per appliance per site.
Hardware development large barrier to entry for new vendors, constraining innovation & competition.
Independent Software Vendors
Virtual Appliance
Orchestrated, automatic & remote install.
Standard High Volume Servers
Standard High Volume Storage
Standard High Volume Ethernet Switches
Network Virtualisation Approach

Figure 2.1: Evolution of NFV from Typical Network Appliances.

The above figure is taken from Source[13]

2.1.2 Relationship with SDN

Network Function Virtualization (NFV) and Software-Defined Networking (SDN) are related concepts but are not interdependent. NFV can be implemented without SDN, and vice versa. However, if we combined these two then can offer additional benefits and create greater value in network management and operations.

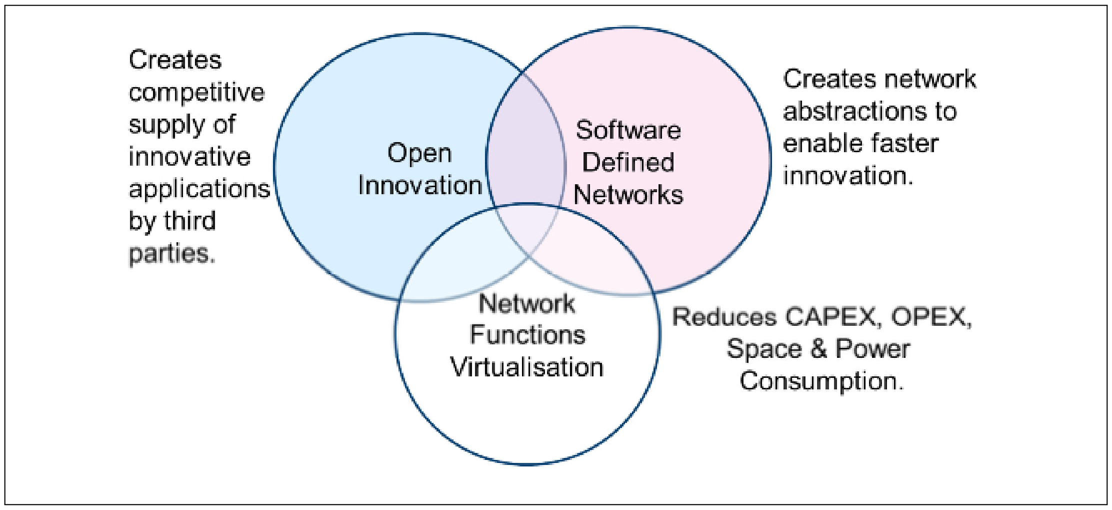

Creates competitive supply of innovative applications by third parties.
Open Innovation
Software Defined Networks
Creates network abstractions to enable faster innovation.
Network Functions Virtualisation
Reduces CAPEX, OPEX, Space & Power Consumption.

Figure 2.2: Network Functions Virtualisation Relationship with SDN

The above figure is taken from Source[13].

Network Function Virtualisation goals can be achieved without using SDN mechanisms.Using the existing technique many data centers achieve NFV goal. However, using SDN utilizing SDN's approach of separating control and data forwarding planes can improve performance, simplify compatibility with existing systems, and make operations and maintenance procedures easier.

2.2 Benefits of NFV

Here we see that some benefit of NFV if a network operator uses NFV to extend its network.

- Using NFV, costs of equipment can be lower than usual and also power consumption can be reduced by mearging equipment and taking advantage of the economies of scale in the IT industry.
- A network operator can be lunch a new product to market more quickly by using NFV. Unlike hardware-based functionalities, software-based development doesn't require economies of scale for investment, which makes it more efficient for operators to develop new features.
- Network operators can run production, test and reference facilities on the same infrastructure if they use NFV. This makes more efficient for testing and integration,which can help reduce development cost and time to market for new products.
- Using NFV, network service can be easily scale up or down as needed. It also speeds up service introduction by remotely supplying software without requiring site visits to install new hardware.
- We can optimizing network setup and structure in almost real time based on actual traffic pattern and service requirement. For example, if we continuously monitoring traffic patterns

and service demand, the network can dynamically allocate resources to where they are needed most and also sift them around in response to change in network demand.

- Supporting multi-tenancy allows network operators to provide services and connectivity for multiple users,applications or internal systems according to their needs using same hardware.each tenants operates with secure separation of administrative domains
- By using NVF, we can use power management features in servers and storage systems. As well as we can reduce workloads and optimize their location.For example, we can focus workloads on fewer servers during times when there is less demand, such as overnight. This way, we can turn off the unused servers and reduce energy consumption.

2.3 Enablers For Network Function Virtualisation

Several recent technology developments have played a significant role in making the goals of Network Functions Virtualization (NFV) achievable. Let's explore some of these enablers and their relevance:

1. **Advancements in Virtualization Technologies:** Advancements in virtualization technologies like hypervisors have made NFV possible. These technologies create virtual machines (VMs) that can host network functions, separating them from dedicated hardware. This separation brings benefits such as flexibility, scalability, and cost-effectiveness in deploying and managing network functions.
2. **Software-Defined Networking (SDN):** SDN plays a vital role as a key enabler for NFV by separating the control and data planes. It creates a centralized and programmable network infrastructure. This programmability allows for dynamic and automated orchestration of virtualized network functions, making network management easier and increasing service flexibility.
3. **Cloud Computing and Infrastructure as a Service (IaaS):** Cloud computing and Infrastructure-as-a-Service (IaaS) platforms have played a crucial role in supporting NFV. Cloud environments provide the necessary on-demand computing, storage, and networking resources required for hosting virtualized network functions. The dynamic nature of NFV deployments aligns well with the scalability and resource elasticity provided by the cloud infrastructure.
4. **Open Standards and APIs:** Open standards and APIs have played a significant role in promoting interoperability and vendor-neutral solutions in NFV. These standards and APIs facilitate the integration and interaction between different virtualized network functions, management systems, and infrastructure components.
5. **Hardware Acceleration and High-Performance Computing:** To ensure network functions perform well, specialized hardware acceleration techniques and high-performance computing technologies have been developed. This ensures that virtualized network functions run efficiently without sacrificing performance.

2.4 Challenges for Network Functions Virtualisation

Along with all the benefits, there are many technical challenges may be face by the network operators when they implementing NFV :-

- **Portability/Interoperability.** The challenge is to create a unified interface that allows virtual appliances to run on different data center environments. This is important for portability and interoperability, which create different ecosystems for virtual appliance and data center vendors. Portability also allows operators to optimize the location and resources of virtual appliances without any restrictions.

- **Performance Trade-Off.** Network Functions Virtualisation (NFV) uses standard hardware, which can decrease performance. The challenges is how to minimize the performance degradation by using appropriate hypervisors and modern software technologies. The available performance of the hardware needs to be clearly indicated so that virtual appliances can use it effectively.

- **Migration and co-existence of legacy & compatibility with existing platforms.** Network Function Virtualisation implementation is done only when it compatible with the network operator's existing equipment and management systems,such as Element Management Systems, Network Management Systems, and OSS/BSS. This compatibility is necessary for the convergence of NFV orchestration and IT orchestration systems. The architecture of NFV must support a migration from proprietary physical network appliances to open standards-based virtual network appliances. This means that NFV must operate in a hybrid network environment that includes both physical and virtual appliances, with virtual appliances using the same management and control interfaces as physical appliances.

- **Automation.** Network Function Virtualisation only applicable when network functions are automated. Automation of process is depends of success.

- **Management and Orchestration.** Having a consistent way of managing and organizing things is important in Network Functions Virtualisation. It allows for flexibility and standardization, which makes it easier and faster to integrate new virtual appliances into a network. Software Defined Networking (SDN) can help by making it easier to control physical switches through virtual appliances or Network Functions Virtualisation orchestration systems.

- **Security & Resilience.** When virtualized network functions are introduced, network operators want to make sure that the security, resilience, and availability of their networks are not compromised. We expect that Network Functions Virtualization (NFV) will enhance network resilience and availability. It allows network functions to be recreated when needed after a failure. If the infrastructure, particularly the hypervisor and its configuration, is secure, a virtual appliance can be just as secure as a physical one. Network operators will look for tools to manage and validate hypervisor configurations. They will also need hypervisors and virtual appliances with security certifications.

- **Network Stability.** When managing and orchestrating a large number of virtual appliances from different hardware vendors and hypervisors, it is important to ensure that the stability of network is not affected. It becomes even more crucial to maintain network stability when virtual functions are relocated or during re-configuration events caused by software/hardware failure or cyber-attacks.

- **Simplicity.** Ensuring virtualized network platforms are expected to be easier to operate compared to current platforms.Network operators are currently focused on simplifying complex network platforms and support systems that have developed over many years, while ensuring they can still support essential revenue-generating services.
- **Integration.** A major challenge in Network Functions Virtualization (NFV) is integrating multiple virtual appliances onto existing industry standard high volume servers and hypervisors seamlessly. Network operators require the flexibility to combine servers, hypervisors, and virtual appliances from different vendors without costly integration and vendor lock-in. The ecosystem should provide integration services, maintenance, and third-party support and offer solutions for resolving integration issues among multiple parties. Validating new Network Functions Virtualization products and developing appropriate tools to address these challenges are also important.

2.5 Architectural Framework

We illustrate the high-level architectural framework of NFV in Figure 2.3. Its four major functional blocks are the orchestrator, VNF manager, virtualization layer, and virtualized infrastructure manager.

The orchestrator plays a crucial role in managing and coordinating various aspects of networking services.It is responsible for the management and orchestration of both software resources and the virtualized hardware infrastructure.

The VNF manager handles tasks like starting, scaling, stopping, and updating a VNF throughout its life cycle. It also supports automation to minimize manual intervention.

The virtualization layer separates the VNFs from the physical resources and connects them to the virtualized infrastructure. It ensures that the VNFs life cycle is independent of the specific hardware used by using standardized interfaces. This is usually done through virtual machines (VMs) and their hypervisors.

The virtualized infrastructure manager is responsible for virtualizing and managing the computer, network, and storage resources. It controls how these resources interact with the VNFs. It assigns virtual machines (VMs) to hypervisors and ensures they are connected to the network. It also identifies and resolves performance issues and collects information about infrastructure faults for planning and optimization purposes.

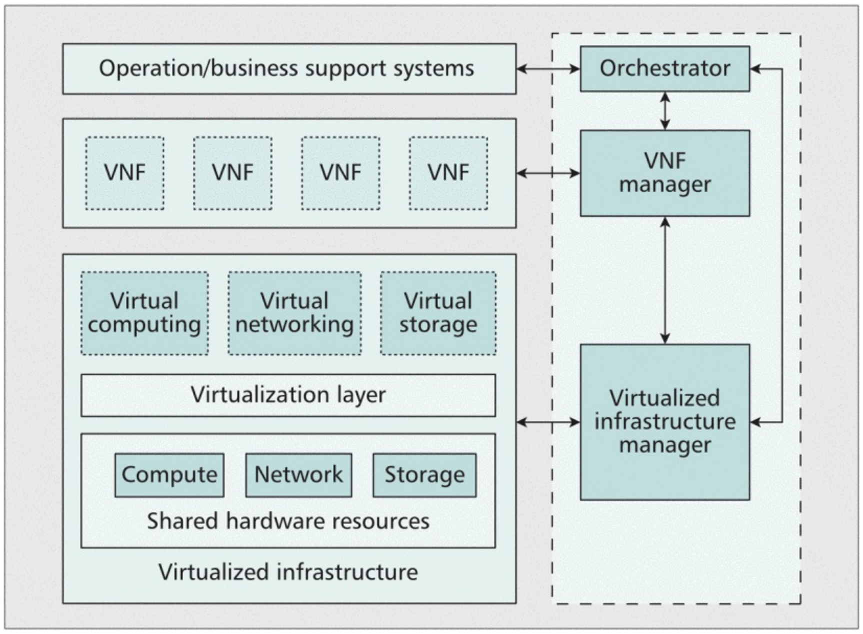


Figure 2.3: NVF Architectural Framework

The above figure is taken from Source[15].

As we can see from this architectural frame-work, the two major things those make NFV possible are industry-standard servers and cloud computing technology. Industry-standard servers are advantageous because they use interchangeable components that are readily available and affordable. In contrast, network appliances use special chips that are custom-made for their specific tasks. These chips can be more expensive and difficult to find compared to the components used in industry-standard servers. By using general-purpose servers, operators can decrease the variety of hardware architectures in their networks. This approach also allows for longer hardware life cycles as technologies evolve (e.g., running different software versions on the same platform). Recent developments of cloud computing, such as various hypervisors, OpenStack, and Open vSwitch, have made NFV a reality. For example, the cloud management and orchestration schemes allow for the automatic creation and migration of virtual machines (VMs) that run specific network services.

2.6 NFV Example Use Cases

2.6.1 Network Virtualization

NFV technologies are mostly being used by the telecom companies around the world for network virtualization. The key feature of NFV - decoupling of hardware and software allows service providers to expand and accelerate the development and innovation of services. Telcos optimize their network services by separating different network functions such as DNS, caching, IDS, and firewall from the proprietary hardware. Network Virtualization gives telecom service providers the agility and flexibility they need when rolling out new network services. It helps them reduce their spending on bulky physical hardware and the costs associated with running, maintaining, and occasionally repairing it.

2.6.2 Mobile Edge Computing

Using network function virtualization allows edge devices to perform computational services and provide network functions by generating and utilizing either a single or multiple virtual machines (VM). Multi-Access Edge Computing (MEC) is a clear example of these technologies. The MEC is utilizing mobile edge computing to provide ultra-low latencies. This technology was born from the ongoing rollouts of 5G networks. The MEC uses individual components in its architecture which are similar to the NFV.

2.6.3 Orchestration Engines

With traditional legacy networks, issues such as low agility, human error, and lack of automatic processes and alerts rendered them limited in their capabilities. NFV orchestration uses automated software to control NFV infrastructure and the VNFs.

2.6.4 Video Analytics

High-performance AI video analysis is mostly performed by cloud-native applications or powerful servers located on the cloud. So, the real challenge is to transfer these large amounts of data for analysis from on-premises to the cloud. Moreover, end-to-end network latency poses a real challenge for the deployment of video analytics.

To mitigate this problem, enterprises have been turning to NFV and SDN architectures to reduce network resource utilization and improve latency. These technologies when combined with video analytics at the network edge, reduce bandwidth use to a great extent.

2.6.5 Security

Using NFV and SDN many security device or component will eventually be virtualized. For example, many security vendors are already offering virtual firewalls to protect VMs. The F5 Gi Firewall VNF Service[1], for example, is one of the most popular NFV solutions that encompasses firewall capabilities.

2.6.6 Network Slicing

Network slicing is dividing a physical network into multiple virtual networks or slices. Each virtual network or slice is fulfill specific requirements of different applications or services and is isolated from other slice to ensure privacy, security and performance. Network slicing has gained a lot of popularity since the beginning of 5G design and rollouts.

[1] https://www.f5.com/cloud/products/distributed-cloud-waf

Chapter 3

NFV Orchestration

3.1 Introduction

NFV (Network Function Virtualization) orchestration refers to the process which automating the management and deployment of network functions on virtualized infrastructure. The goal of NFV orchestration is to simplify the complex task of managing network functions by providing a centralized platform for managing the entire virtualized network.

NFV orchestration involves automating the entire life cycle of network functions, from deployment and configuration to scaling and shut off. This automation is achieved through the use of software-defined networking (SDN) and network automation tools, which enable network operators to manage and deploy network functions in a more flexible and scalable way.

NFV orchestration helps network operators work faster, more efficiently and at a lower cost. By using automation to manage network functions, they can set up new services and applications more quickly and easily. They can also change the size of their network faster if they need to, without having to do it all manually. This means they can save time and resources, and respond better to changes in demand.

3.2 What is Orchestration?

Orchestration involves automating the configuration, management, and coordination of computer systems, applications and services. It helps IT teams handle complex tasks and workflows more easily. [17]

Automation and orchestration are related concepts but have different scopes. Automation focuses on automating individual tasks, while orchestration automates processes or workflows that involve multiple steps across different systems.

Orchestration enables the streamlining and optimization of recurring processes and workflows, supporting a DevOps approach and speeding up application deployment. It improves the efficiency of IT processes such as server provisioning, incident management, cloud orchestration, database management, and application orchestration.

3.3 Why NFV Orchestration?

NFV (Network Function Virtualization) is a modern networking approach that helps deploy network functions quickly and efficiently. It reduces costs and improves flexibility by using software-based functions instead of dedicated hardware.To do that, NFV orchestration is needed to dynamically coordinate the underlying infrastructure resources.Therefore, NFV orchestration is crucial for managing the entire life cycle of Virtual Network Functions (VNFs) and end-to-end network services. It handles tasks like creating, configuring, and ending VNFs, adjusting their scale as needed, managing resources and policies, measuring performance, correlating events, and validating and authorizing resource requests. In simple terms, NFV orchestration ensures smooth operation and control of virtual network functions and services from start to finish.And also we can combine NFV, SDN, and clouds for automatically set up and manage network services. This is done by deploying virtualized network functions (VNFs) that make up complex network services and directing traffic between them.

3.4 Overview of NFV Orchestrators

An orchestrator[17] is a software((or set of software components) responsible to deploy VNFs in shared hosting infrastructures and combine them with other services, such as physical network functions (PNFs), to create advanced network services for applications and users.According to ETSI, in a NFV environment the following distinction must be considered:

- **Resource Orchestrator:** The Resource orchestrator manages and controls the allocation and utilization of network resources across different locations or within a single location.
- **Service Orchestrator:** The Service orchestrator solves the problem of connecting different virtual network functions (VNFs) that may be managed by different VNF Managers. It creates end-to-end services between these VNFs, ensuring they work together seamlessly.

While NFV orchestration has gained attention from industry and academia, more work is needed to develop complete NFV management and orchestration (MANO) solutions that meet the requirements of service and network providers. For example the industrial project Weaver, a VNF manager for multi-domain and multi-vendor VNFs orchestration and lifecycle management, proposed by Openet3 does not consider VNF chaining. Weaver does not use any SDN technology for flow management either and supports only the OpenStack virtualized infrastructure manager (VIM).

The OpenStack organization introduced an NFV orchestration project called Tacker for VNFs deployment on an OpenStack-based NFV platform. Other open source tools like OPNFV, Open Baton, OPEN-O, and OpenMANO also address NFV orchestration. OPNFV is an open source project that was initially focused on building NFV infrastructure (NFVI) and virtualized infrastructure management. It has been recently extended to include MANO components for application composition and management. Open Baton is a framework for VNF MANO for emerging software-based fifth generation (5G) networks that runs on top of multi-site OpenStack clouds. OPEN-O is also an open source project launched by the Linux Foundation that implements ETSI's NFV-MANO.

3.5 VNF Orchestration Framework

According to the ETSI, MANO architecture breaks down the NFV management and orchestration architecture into three functional lavers:

- VIMs
- VNF managers (VNFMs)
- NFV orchestrator (NFVO)

Figure 3.1 shows the important features that NFVO (NFV Orchestrator) and VNFM (Virtual Network Function Manager) should have. The ETSI MANO (European Telecommunications Standards Institute Management and Orchestration) architecture is a framework designed to manage and orchestrate virtualized network functions (VNFs) in a cloud-based network environment. It provides a standardized approach to managing the lifecycle of VNFs, including their instantiation, scaling, and termination.

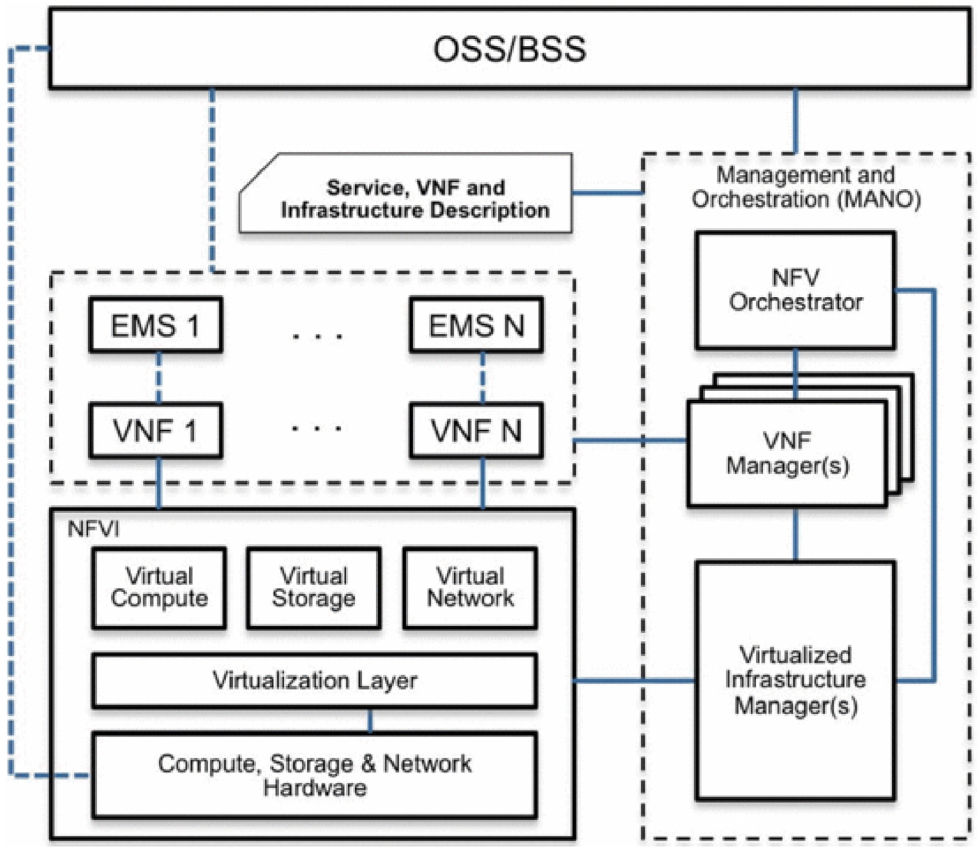

OSS/BSS
Service, VNF and Infrastructure Description
Management and Orchestration (MANO)
NFV Orchestrator
EMS 1
. . .
EMS N
VNF 1
. . .
VNF N
VNF Manager(s)
NFVI
Virtual Compute
Virtual Storage
Virtual Network
Virtualization Layer
Compute, Storage & Network Hardware
Virtualized Infrastructure Manager(s)

Figure 3.1: ETSI MANO Architectural Framework

The above figure is taken from Source[16]

Summary of notations:

- **NVF** is the virtualized network element like Router VNF, Switch VNF, Firewall etc.
- **NFVI Resources :** A repository of NFVI resources utilized for the purpose of establishing NFV services.
- **VIM :** Virtualized Infrastructure Manager (VIM), manages NFVI resources in one domain.
- **VNFM :** VNF Manager(VNFM) Manages life cycle of VNFs. It creates, maintains and terminates VNF instances, installed on VMs which the VIM creates and manage

However, it's important to note that the ETSI MANO architecture focuses on defining the building blocks and functionalities of the system rather than specifying the technical details of how the system should be implemented from end to end. In other words, it outlines the high-level concepts and requirements but does not provide specific guidelines or recommendations for the technical implementation.

The reason behind this approach is to allow flexibility and adaptability in the implementation of MANO systems. Different organizations and vendors may have different technical requirements, preferences, and existing infrastructures, so the ETSI MANO architecture intentionally avoids prescribing a specific technical approach. This enables organizations to choose and implement the most suitable technologies, protocols, and mechanisms that align with their specific needs and constraints.

By providing a standardized set of functionalities and interfaces, the ETSI MANO architecture ensures suitable between different components and vendors within the overall MANO ecosystem. This allows for the integration of various VNFs from different vendors, enabling multi-vendor environments and promoting competition and innovation in the market.

3.6 NFV Orchestration Requirements

As virtualized networks span over a large number of networks, software elements, and hardware platforms, NFV orchestration tools must be powerful enough to work with many different operating standards and/or principles. Some of the functions typically required by NFV orchestration include:

- **Service coordination and instantiation**: The orchestration software must communicate with the underlying NFV platform to instantiate a service, which means it creates the virtual instance of a service on the platform.
- **Service chaining**: Enables a service to be cloned and multiplied to scale for either a single customer or many customers.
- **Scaling services**: When more services are added, the orchestration software must find and manage sufficient resources to deliver the service.
- **Service monitoring**: Tracks the performance of the platform and resources to make sure they are adequate to provide good service.

Chapter 4

Tools

4.1 Open Source NFV Orchestration Platform

4.1.1 Open Source MANO (OSM)

The OpenSource MANO[5] project was officially launched at the World Mobile Congress (WMC) in 2016. Starting with several founding members, including Mirantis, Telefónica, BT, Canonical, Intel, RIFT.io, Telekom Austria Group and Telenor, the OSM community now includes 55 different organisations. The OSM project is hosted at ETSI facilities and targets delivering an open source management and orchestration (MANO) stack closely aligned with the ETSI NFV reference architecture. The current release Release THIRTEEN introduces a new scalable architecture for service assurance and closed-loop operations leveraging on cloud-native version of Apache Airflow and Prometheus.

4.1.2 OPEN-O

The OPEN-O[4] project is a comprehensive platform for orchestration, management, and automation of network and edge computing services for network operators, cloud providers, and enterprises. Real-time, policy-driven orchestration and automation of physical and virtual network functions enables rapid automation of new services and complete lifecycle management critical for 5G and next-generation networks. The OPEN-O project is hosted by the Linux foundation and was also formally announced at 2016 MWC.

4.1.3 CORD

The CORD[1] (Central Office Re-architected as a Datacenter) platform leverages SDN, NFV and Cloud technologies to build agile datacenters for the network edge. Integrating multiple open source projects, CORD delivers a cloud-native, open, programmable, agile platform for network operators to create innovative services. CORD provides a complete integrated platform, integrating everything needed to create a complete operational edge datacenter with built-in service capabilities, all built on commodity hardware using the latest in cloud-native design principles.

4.1.4 Cloudify

Cloudify[2] is an open-source multi-cloud and edge orchestration platform. Cloudify allows organizations an effortless transition to public cloud and Cloud-Native architecture by enabling them to automate their existing infrastructure alongside cloud native and distributed edge resources. Cloudify also allows users to manage different orchestration and automation domains as part of one common CI/CD pipeline. Key Features Everything as a Code Service Composition Domain-Specific Language (DSL) - enabling modeling of a composite service, containing components from multiple Cloudify services and other orchestration domains.

4.2 Mininet

Mininet [3] is a widely used network emulator that relies on process-level virtualization. This type of lightweight virtualization emulates guest machines as isolated processes, reserving memory, CPU and network, inheriting the host functions and programs, and enabling the design of large-scale network environments.

SDN-enabled network topology of our experiment is emulated using Mininet 2.2.2 with Open vSwitch (OVS) 2.5.4 supported by OpenFlow version 1.3. Mininet can be used as both an emulator and a simulator. It is widely used by many researchers for performance evaluation of SDN-based networks in different domains such as acoustic sensor networks, communication networks, and security.

A management problem of the physical resources such as CPU power and memory can lead to bottlenecks in the Mininet environment, especially in large-scale emulated networks [18]. In our experiments, we use small-size topologies. Furthermore, we employ the flexibility of Mininet to allocate fractions of the resources of the host machine such as CPU power, memory allocation, etc. For example, we manage CPU power through "mininet.node.CPULimitedHost Class" to assign a fraction of CPU power for each emulated host node (i.e., VM) . Similarly, the characteristic parameters of each virtual link such as bandwidth, packet loss, delay, and jitter can be assigned a specific value. Mininet also creates unique network namespaces to isolate the communications between the VMs and the host system. Therefore, our experiments are conducted in a dedicated, uninterrupted, and stable platform.

4.3 Mini-NFV

Mini-nfv [12, 11] is a framework for NFV Orchestration with a general purpose VNF Manager to deploy and operate Virtual Network Functions (VNFs) and Network Services on Mininet. It is based on ETSI MANO Architectural Framework. Mini-nfv manages the life-cycle of a Virtual Network Function (VNF). Mini-nfv takes care of deployment, monitoring, scaling and removal of VNFs on Mininet.

Mini-nfv[1] allows loading OASIS TOSCA templates (V1.0 CSD 03) into Mininet.

[1]https://github.com/josecastillolema/mini-nfv

4.4 OpenStack

OpenStack[6] is an open-source cloud computing platform that provides a framework for building and managing private and public cloud environments. It is developed by NASA and RackSpace, OpenStack comes under open source software category and it is freely available to download and use. It controls large number of compute, storage, and networking resources throughout a data center. It allows these resources to be easily accessed and provisioned through APIs with common authentication mechanisms.

With OpenStack, you can easily scale your resources, automate tasks, and deploy applications, making it an ideal solution for organizations of all sizes. Whether you're a developer, system administrator, or IT professional, OpenStack offers the flexibility and control you need to effectively manage your cloud environment.

A dashboard is also available, which give administrators control to users to provision resources through a web interface.

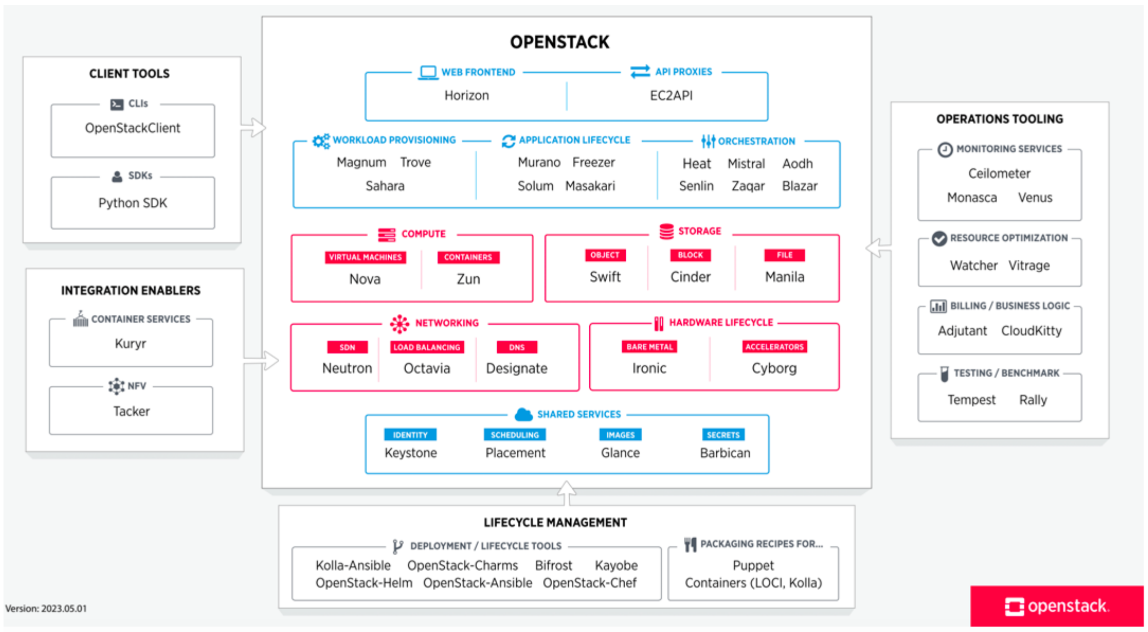


Figure 4.1: OpenStack Components

The detail information about OpenStack describe in **Chapter5.**

4.5 Tacker

Tacker[8] is an OpenStack service for NFV Orchestration with a general purpose VNF Manager to deploy and operate Virtual Network Functions (VNFs) and Network Services on an NFV Platform. It is based on ETSI MANO Architectural Framework.

4.6 TOSCA

The Topology and Orchestration Specification for Cloud Applications (TOSCA[9] is an open source language used to describe the relationships and dependencies between services and applications that reside on a cloud computing platform.

TOSCA can describe a cloud computing service and its components, and document the way those components are organized as well as the orchestration process needed to use or modify those components and services. This provides administrators with a common way to manage cloud applications and services so that they can be portable across different cloud vendors' platforms.

The TOSCA language describes cloud services using templates and plans. Templates define the structure of a cloud service. Plans define the processes that start, stop and manage that cloud service over its lifetime. For example, TOSCA could be used to describe the relationship between Docker containers, virtual machines, server components, endpoints and services within a cloud environment. This enables faster, repeatable and scalable application deployments.

TOSCA is a highly extensible language, enabling developers to add vendor- or domain-specific mechanisms to accommodate specific use cases. For example, a cloud provider could use TOSCA to define and compose a specific cloud service.

TOSCA facilitates simpler application deployment to any cloud platform, cloud bursting and support for multi-cloud environments. TOSCA can also facilitate the standardization of cloud-based services, which enables cloud providers to offer ubiquitous services that users can map to their respective infrastructures.

Chapter 5

Open Stack

5.1 Overview of OpenStack

OpenStack[6] is a widely adopted open source cloud operating system framework.Since its first release in 2010, OpenStack has gained popularity and maturity over the years, thanks to the collaborative efforts of numerous developers and users. Currently, OpenStack is a versatile and widely adopted platform utilized in various domains such as private cloud, public cloud, and NFV (Network Functions Virtualization).

5.1.1 OpenStack Architecture

A cloud operating system is a distributed system that includes software and hardware components. Just like a regular operating system, it requires management. OpenStack is a crucial element in creating a cloud operating system, particularly for deploying infrastructure as a service (IaaS) or building a comprehensive framework for cloud operations. **Fig 5.1** shows the cloud operating system in the cloud computing framework, and we can see OpenStack's position in cloud computing framework.

The cloud operating system framework is not equal to the cloud operating system. Building a complete Cloud OS involves integrating software components that work together to provide the functionality and services that system administrators and tenants need. On the other hand, OpenStack alone does not have certain essential capabilities for a full-fledged cloud operating system. For example, OpenStack cannot handle resource access and abstraction on its own. It relies on underlying virtualization software, software definition storage, software definition network, and other software to fulfill this functionality. Similarly, it requires integration with various management software platforms for comprehensive application life cycle management. OpenStack also doesn't possess complete system management and maintenance capabilities, so it requires the integration of diverse management software and maintenance tools when deployed.

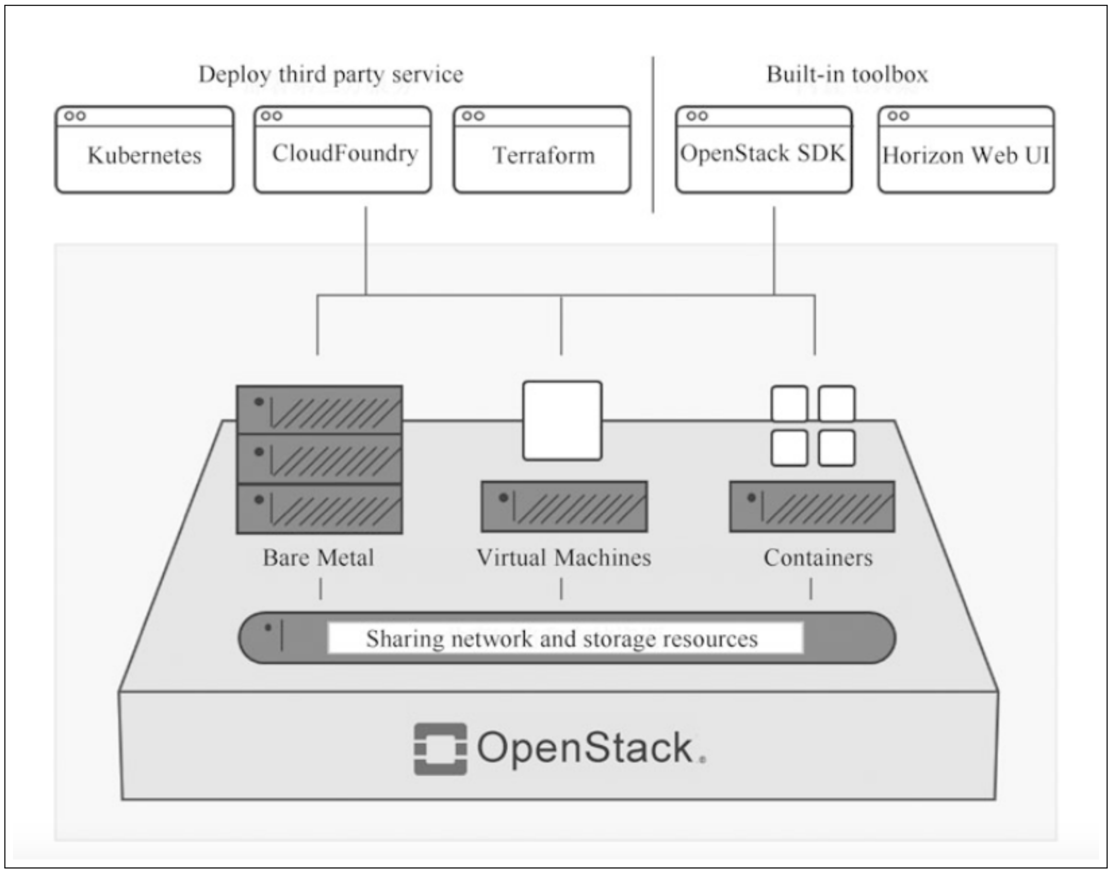


Figure 5.1: Cloud operating system in the cloud computing framework (Source[6])

It is evident how building a comprehensive cloud operating system using OpenStack requires integrating it with other software components to fill the gaps in its capabilities. For this OpenStack serves as a framework for creating cloud operating systems by combining various components to meet specific requirements. Using this framework, various components can be integrated to implement cloud operating systems that meet the needs of different scenarios. By doing so, it becomes possible to build a complete cloud computing system.

5.1.2 OpenStack Core Components

OpenStack contains many components, including Nova, Swift, Glance, Keystone, Neutron, Cinder, Horizon, MQ, Heat, Ceilometer, and seven core components.

1. **Nova:** Nova is OpenStack's controller that handles all the necessary tasks for managing instances in the OpenStack cloud throughout their life cycle. It takes care of computing resource management and scaling needs. However, Nova uses Libvirt's API to facilitate interactions between hypervisors, as it doesn't provide virtualization capabilities of its own.

2. **Swift:** Swift delivers object storage services within OpenStack, enabling the storage and retrieval of files without the need for mounting directories on the file server. It empowers OpenStack with a distributed and ultimately consistent virtual object storage solution. With its distributed storage nodes, Swift can handle vast numbers of objects. It also integrates built-in redundancy, fault management, archiving, and streaming capabilities, making it highly scalable. With its distributed, ultimately consistent virtual object storage, Swift complements other OpenStack components like Nova (compute) and Glance (image service) by providing a reliable and scalable storage solution.

3. **Glance:** Glance serves as a storage repository and directory for virtual disk images that enabling the storage and retrieval of virtual machine images. These disk images are extensively utilized in Nova components. Glance supports mirror management and tenant private mirror management across multiple data centers.

4. **Keystone:** Keystone is responsible for authentication and authorization across all services in OpenStack. Given the complexity involved in authentication and authorization, particularly in large projects like OpenStack, Keystone ensures unified authentication and authorization for each component.

5. **Neutron:** Neutron is the core component for delivering network services in OpenStack. It operates on the concept of a software-defined network and manages network resources through software. It allows users to create virtual networks, routers, subnets, and ports. Neutron maximizes the utilization of various network technologies in the Linux operating system and supports third-party plug-ins.

6. **Cinder:** Cinder is a crucial component of the virtual infrastructure that handles the storage of disk files and data used by virtual machines. It offers block storage services specifically for instances. Storage allocation and usage are managed through block storage drives or multiple configured drives.

7. **Horizon:** Horizon is a user-friendly, web-based interface designed to facilitate the management of various OpenStack resources and services. It serves as a central hub for cloud administrators and users, allowing them to easily access and control different aspects of their OpenStack environment. With Horizon, users can efficiently manage virtual machines, networks, storage, and other components, simplifying the administration and utilization of OpenStack resources.

5.1.3 Logical Relationship Between OpenStack Components

OpenStack services are accessed using a unified REST-style API to achieve loose coupling of the system. This loosely coupled architectural approach enables individual component developers to focus on their specific areas without affecting others. However, it also poses challenges for system maintenance as operations personnel need to master more system-related knowledge to troubleshoot component issues. Therefore, both developers and maintenance personnel must understand the interactions between components to effectively manage the system.

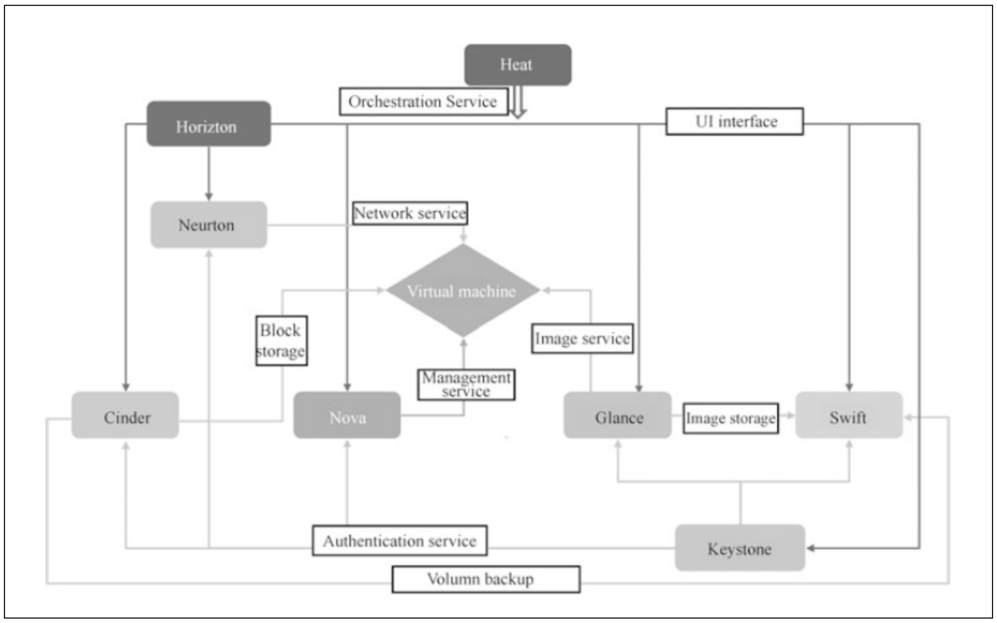


Figure 5.2: The logical relationship between OpenStack components (Source[6])

Figure 5.2 shows the logical relationship between the various components of the virtual machine. Here are some key logical relationships between OpenStack components:

1. **Nova and Cinder:** Nova, the compute component, relies on Cinder, the block storage component, to provide persistent storage for instances. Nova interacts with Cinder to attach, detach, and manage block storage volumes used by virtual machines.

2. **Nova and Neutron:** Nova and Neutron, the networking component, work together to enable network connectivity for instances. Nova uses Neutron's networking capabilities to assign IP addresses, configure network interfaces, and establish network connectivity for virtual machines.

3. **Glance and Nova:** Glance, the image service component, integrates with Nova to provide virtual machine images for instance creation. Nova utilizes Glance to retrieve, store, and manage virtual machine images used as templates for launching instances.

4. **Keystone and All Components:** Keystone, the identity service component, serves as the central authentication and authorization system for OpenStack. All other components, including Nova, Cinder, Neutron, and others, rely on Keystone for user authentication and authorization.

5. **Horizon and All Components:** Horizon, the OpenStack dashboard, provides a web-based graphical user interface (GUI) for managing and interacting with OpenStack services. It acts as a front-end interface to access and control various components of OpenStack, including Nova, Cinder, Neutron, and more.

These logical relationships enable seamless coordination and interaction between different components, allowing for the creation, management, and operation of a comprehensive cloud infrastructure in OpenStack.

5.2 OpenStack Operating Interface Management

OpenStack provides an operating interface management system that allows users to interact with and manage various components of the OpenStack cloud infrastructure. OpenStack includes a user-friendly interface called Horizon, which serves as the operator panel. It allows end-users and developers to easily navigate and manage computing resources within OpenStack.

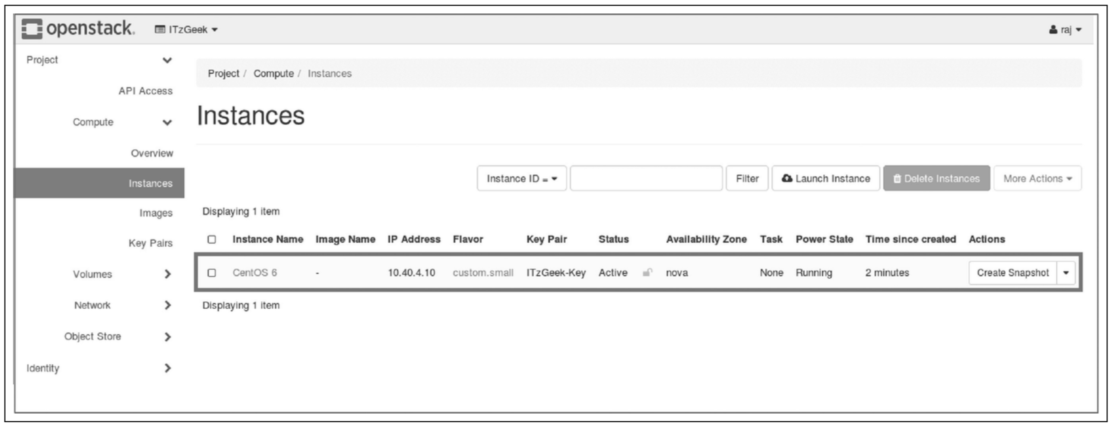
5.2.1 Introduction to OpenStack Operation Interface

The OpenStack dashboard, also known as Horizon, is a web-based graphical user interface (GUI). It is the main gateway to the OpenStack application architecture. It offers a a user-friendly interface where users can access and manage their compute, storage, and network resources. Through the Horizon portal, users can perform tasks like launching virtual machine instances, assigning IP addresses, and managing access controls, all using a browser-based interface. Horizon offers distinct interfaces for users with different roles.

- **Cloud administrator:** Horizon allows cloud administrators to have a comprehensive view of the cloud environment. They perform various tasks related to the administration and maintenance of the OpenStack environment. They can monitor resource size and health, create end-users and projects, assign projects to end-users, and manage resource quotas for projects.
- **End-users:** Horizon offers end-users an autonomous service portal where they can access and utilize compute, storage, and network resources within the projects assigned to them by cloud administrators. The portal ensures that end-users stay within the defined quota limits set by the administrators while benefiting from the available resources. This empowers end-users to independently manage and leverage the resources they need for their specific projects within the OpenStack environment.

5.2.2 The Architecture and Functions of the OpenStack Operation Interface

Horizon relies on the Django framework, a widely adopted open-source web application framework built with Python. By following the pattern of Django, Horizon generates multiple apps that work together to deliver a fully functional implementation of the OpenStack interface. This approach ensures that Horizon leverages the robustness and flexibility of Django to provide a seamless and user-friendly experience for managing OpenStack resources through a web-based interface. **Figure 5.3** shows the interface in which Horizon created the instance.



Displaying 1 item

Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Time since created	Actions
CentOS 6	-	10.40.4.10	custom.small	ITzGeek-Key	Active	nova	None	Running	2 minutes	Create Snapshot

Displaying 1 item

Figure 5.3: Horizon interface (Source[6])

Horizon is composed of three main dashboards, also known as Django apps: the User Dashboard, the System Dashboard, and the Set Dashboard. Together, these three dashboards form the core application of Horizon, providing different functionalities and perspectives for users and administrators.

1. **User Dashboard:** The User Dashboard is designed for end-users or regular users of the OpenStack cloud. It provides a user-friendly interface for managing and controlling resources specific to individual users or projects. Within the User Dashboard, users can perform tasks such as launching instances (virtual machines), managing volumes, configuring networks, and accessing their personal settings. This dashboard focuses on empowering users to interact with their allocated resources and perform self-service operations within their designated scope.

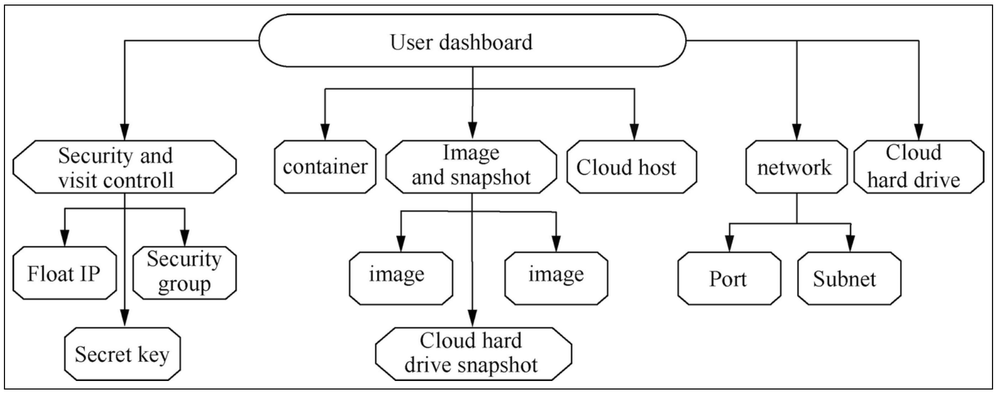


Figure 5.4: Functional architecture of User Dashboard (Source[6])

2. **System Dashboard:** The System Dashboard is primarily targeted towards administrators or operators responsible for managing the overall OpenStack infrastructure. It offers a comprehensive set of tools and features for monitoring and controlling the system-level operations.

Administrators can configure global settings, manage system-wide resources, set up authentication and authorization policies, and monitor the health and performance of the OpenStack environment. The System Dashboard provides a centralized interface for managing the underlying infrastructure and governing the overall operation of the cloud.

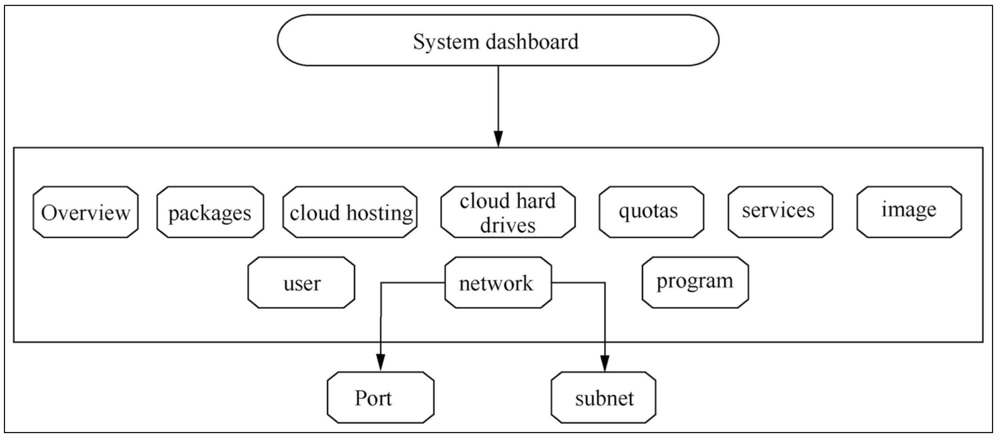


Figure 5.5: Functional architecture of the System Dashboard (Souece[6])

3. **Set Dashboard:** The Set Dashboard is a customizable dashboard that allows users or administrators to create their own custom dashboards tailored to their specific needs. It enables users to assemble and arrange widgets, which are small applications or tools that provide specific functionality or display specific information. Users can select and arrange widgets from a set of available options to create personalized dashboards that align with their preferences and requirements. The Set Dashboard promotes flexibility and customization, empowering users to create a personalized view of their OpenStack resources and services.

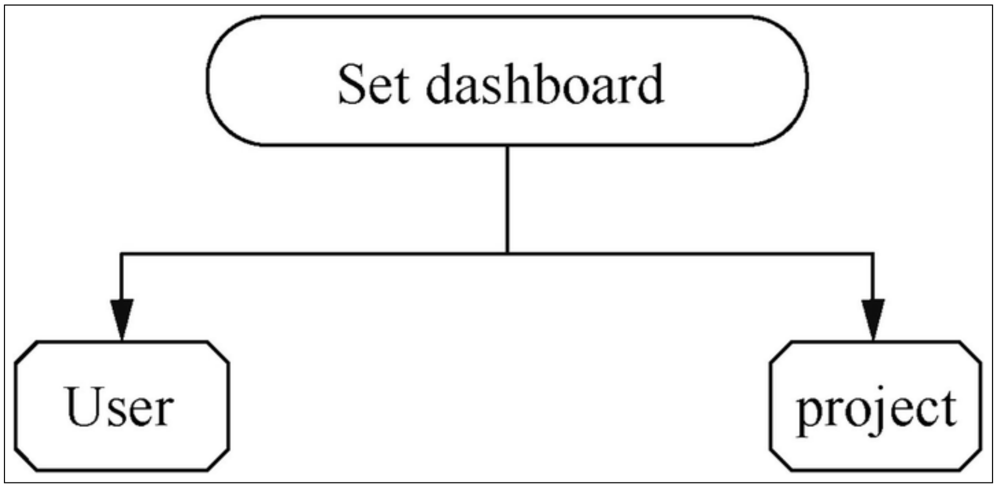


Figure 5.6: Setting the functional architecture of Set Dashboard (Source[6])

Apart from its primary web interface features, Horizon offers additional control over various page details through different options available in the profile. For example, administrators can customize the OpenStack home page by setting a logo image using the local.setting.py file in the profile. They can also specify the title of the page to further personalize the Horizon interface according to their preferences. These options allow for a more customized user experience within the Horizon portal.

5.3 OpenStack Certification Management

Security is a crucial aspect that cannot be ignored in any software.It must be addressed in every software, including OpenStack. It is essential to consider security considerations and implement, although no software can completely resolve all security issues. OpenStack, being a provider of cloud infrastructure services, places significant emphasis on security and privacy to ensure the trust and confidence of its users.

5.3.1 Introduction to OpenStack Authentication Service

Keystone is a component of the OpenStack framework responsible for managing authentication, service access rules, and service tokens. It serves as a centralized identity management system, enabling users to securely access and interact with various OpenStack services. Keystone is similar to a service bus or registry of the entire OpenStack framework. Each OpenStack service registers its access URL, called the Endpoint, with Keystone. Any communication between services requires authentication by Keystone, obtaining the target service's Endpoint, and then making the necessary calls.

Keystone's main features are as follows:

- **Identity Management:** Keystone manages user identities, including user accounts, groups, and roles, within the OpenStack environment. It acts as a single source of truth for user authentication and authorization across the entire OpenStack infrastructure. Keystone supports multiple authentication methods, including username/password, token-based authentication, and integration with external identity providers such as LDAP or Active Directory.

- **Service Catalog:** Keystone maintains a service catalog that provides a centralized repository of available OpenStack services and their endpoints. It allows users to discover and access the various services provided by OpenStack, such as Nova (compute), Cinder (block storage), Neutron (networking), and more. The service catalog helps users navigate and interact with different OpenStack services effectively.

- **Certification and Authorization:** Keystone verifies the identity of users and services attempting to access the OpenStack resources. It authenticates users based on their credentials and grants them access to the appropriate resources based on their assigned roles and permissions. Once a user is authenticated, Keystone performs authorization to determine the actions and resources that the user is allowed to access within the OpenStack environment. Keystone ensures that users can only perform actions and access resources that align with their assigned roles and permissions.

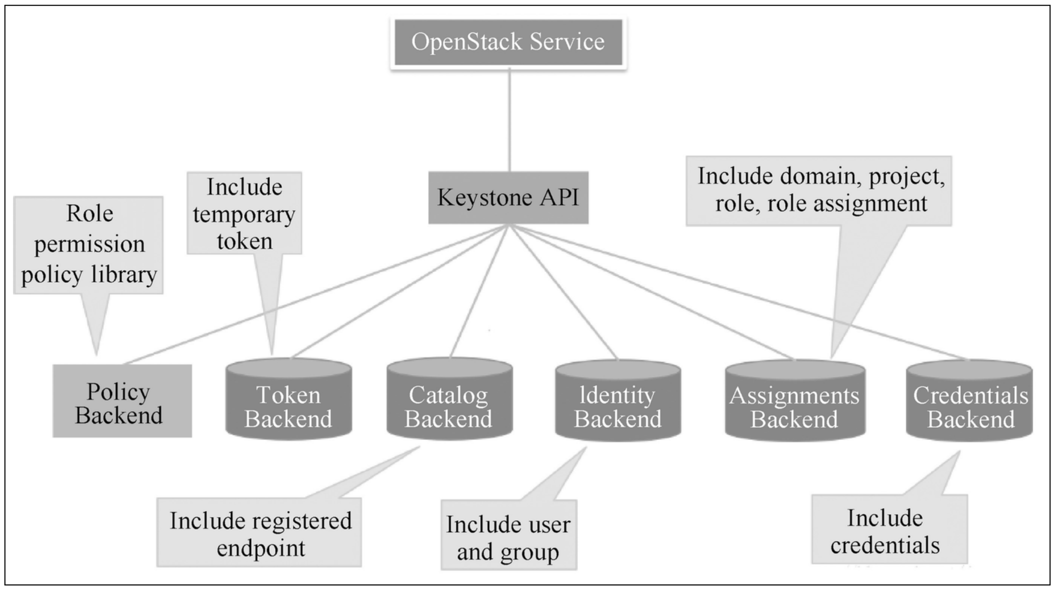


Figure 5.7: Keystone architecture (Source[6])

5.3.2 Principles of OpenStack Authentication Service

Keystone, as a standalone security authentication module in OpenStack, handles important tasks such as user authentication, token management, and service directories. It ensures that users can access resources based on their roles and provides access control. Keystone is responsible for verifying usernames and passwords, issuing tokens, registering services (Endpoints), and determining whether a user has access to specific resources. It plays a critical role in managing user authentication and access within the OpenStack system.

Based on the core concepts, Keystone provides services in four areas:

1. **Identity:** During the authentication process, the user's identity is confirmed by providing their username and password. The Identity service then extracts relevant metadata associated with that user.

2. **Token:** After the user's identity is confirmed, they are provided with a token that serves as a verification of their identity. This token can be used to request subsequent resources. The Token service in Keystone verifies and manages these tokens. Keystone issues two types of tokens to users through the Identity service. One type is a general token that is not tied to any specific tenant, allowing users to browse the list of available tenants in Keystone. The user can then select a specific tenant and obtain another type of token that is bound to that tenant. This tenant-bound token enables access to resources limited to that particular tenant. Tokens have a limited lifespan and can be revoked if necessary, such as when removing a user's access.

3. **Catalog:** The Catalog in Keystone offers a convenient way to query the service catalog or obtain a list of endpoints for each service. It contains endpoint information for all the services

available. When accessing resources between services, the starting point is the endpoint information of that particular resource. This usually consists of a list of URLs that provide access to the resource. In the current version, Keystone provides the service directory along with the token, enabling users to access the directory simultaneously when they receive the token.

4. **Policy:** The authentication engine in Keystone operates based on rules defined in configuration files, which determine how actions align with user roles. It's worth noting that this aspect is no longer included in the Keystone project itself, as access control is now managed separately within various projects. Therefore, the development and maintenance of this component are handled as part of online and offline interactions.

Keystone serves as a crucial bridge between users and services within the OpenStack framework. It facilitates the interaction by allowing users to obtain a token and service list from Keystone. When a user accesses a particular service, they send their token along with the request. The service, in turn, verifies the authenticity of the token by consulting Keystone. This process ensures that users can securely access the services they are authorized to use, creating a seamless and secure connection between users and services in the OpenStack environment. **Figure 5.8** shows the service user interaction process based on the Keystone mechanism.

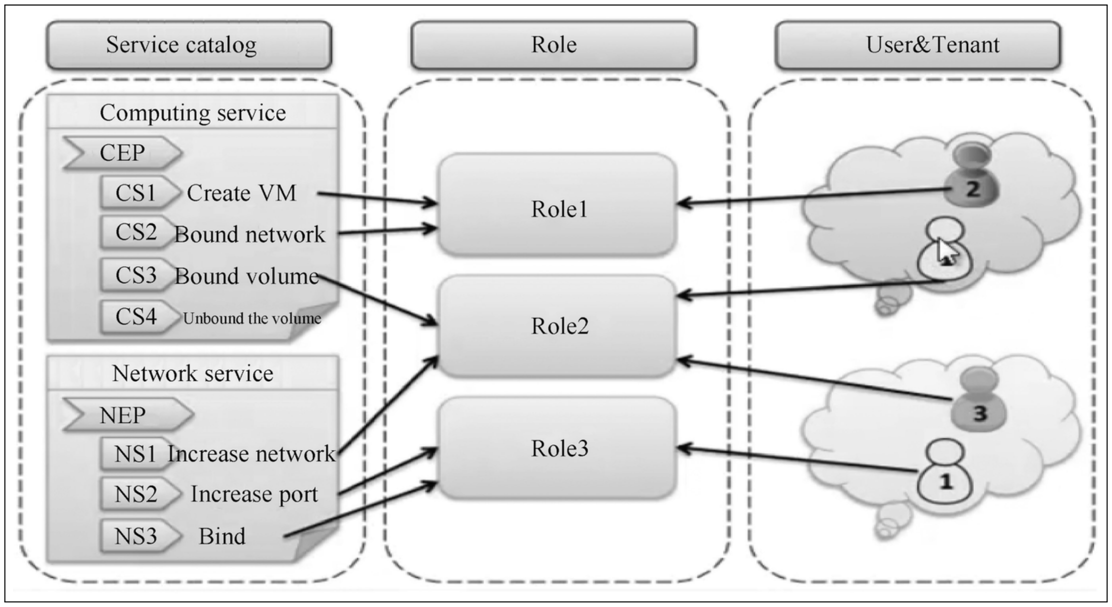


Figure 5.8: Service user interaction process based on the Keystone mechanism (Source[6])

5.4 OpenStack Image Management

OpenStack Image Management refers to the process of handling virtual machine images within the OpenStack cloud platform. Mirroring plays a crucial role in cloud and virtualization environments.

This section provides an overview of OpenStack mirror management by describing the Glance components. Glance serves as a directory and storage repository for virtual disk images, enabling the storage and retrieval of virtual machine images. These disk images are commonly utilized by OpenStack's Nova-compute components for various operations.

5.4.1 Introduction to OpenStack Image Service

Glance serves as a mirroring service for virtual machines in OpenStack, offering a set of REST APIs for image management and querying. It supports multiple backend storage options, including the use of local file systems or Swift for storage. **Figure 5.9** describes Glance's relationship with Nova and Swift.

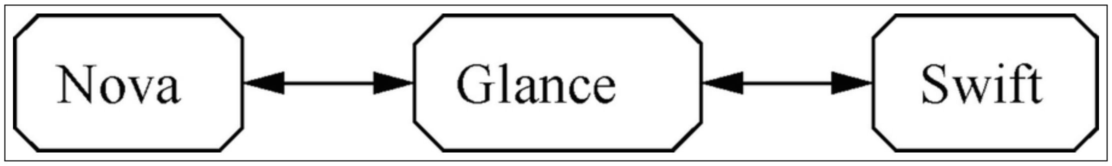


Figure 5.9: The relationship between Glance, Nova, and Swift (Source[6])

As you can see, Glance, Nova, and Swift are interconnected components in OpenStack. Glance acts as a bridge between Nova and Swift. It helps Nova find and retrieve images, while Swift serves as the storage platform for Glance. Swift implements the storage interface required by Glance, providing the actual storage services needed for image management. In this way, Glance, Nova, and Swift work together seamlessly to handle image storage and retrieval within the OpenStack environment.

5.4.2 Principles of OpenStack Image Service

The OpenStack mirroring service, Glance, consists of two main components: the API Server and the Registry Server. Glance is designed to be flexible, supporting various back-end storage and registration database setups. he API Server, powered by the "Glance-API" program, acts as a central hub for communication between client programs, mirror metadata registration, and storage systems. It handles client requests by forwarding them to the mirror metadata registry and the storage system. This allows Glance to effectively store virtual machine images using these mechanisms.

The Glance-API component is responsible for handling API requests and performing corresponding actions. On the other hand, Glance-registry interacts with MySQL databases to store and retrieve metadata related to mirroring. It's important to note that Swift, the storage service, does not store this metadata. Instead, it is stored in the MySQL database and belongs to the Glance component.

The process of creating a virtual machine instance is carried out by the Nova-compute component, which has a strong connection with Glance. These two components work closely together to ensure the successful creation and operation of virtual instances in the OpenStack environment. The process for creating an instance is shown in **Figure 5.10**

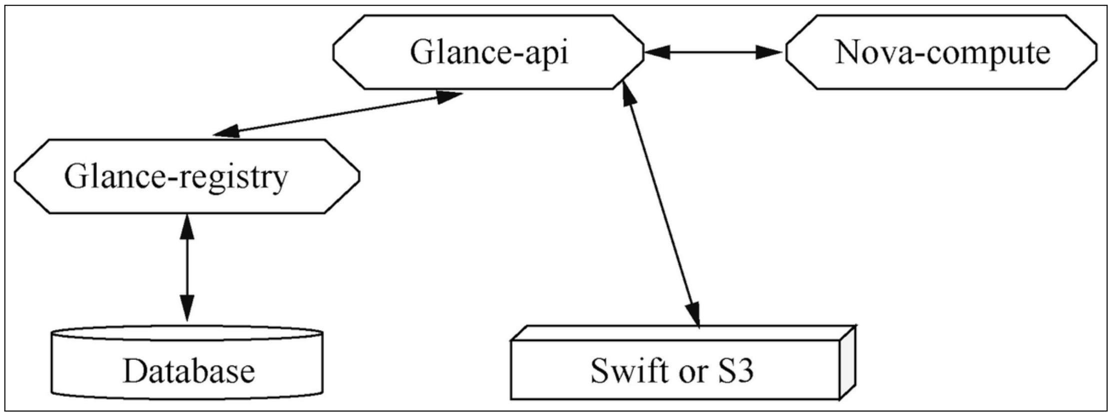


Figure 5.10: The process of creating an instance (Source[6])

5.5 OpenStack Computing Management

Nova plays a vital role in the OpenStack ecosystem as the core component responsible for managing and overseeing the computing resources within a cloud infrastructure. It handles the complete life cycle management of virtual machines.Also providing essential functionalities for creating, running, and managing instances in an Infrastructure-as-a-Service (IaaS) environment.

5.5.1 Introduction to OpenStack Computing Service

At the core of OpenStack is the computing organization controller, which delivers flexible and scalable computing resources on demand. Nova, the component responsible for managing the life cycle of instances.It plays a crucial role in handling all activities within the OpenStack cloud. With its scalability, Nova serves as a robust platform for effectively managing computing resources.

In the earlier versions of OpenStack, Nova handled computing, storage, and networking. However, as OpenStack evolved, the storage and networking functionalities were separated. Today, Nova focuses specifically on computing services. It relies on Keystone for authentication, Neutron for networking, and Glance for image mirroring services.The Nova architecture is shown in **Figure 5.11**

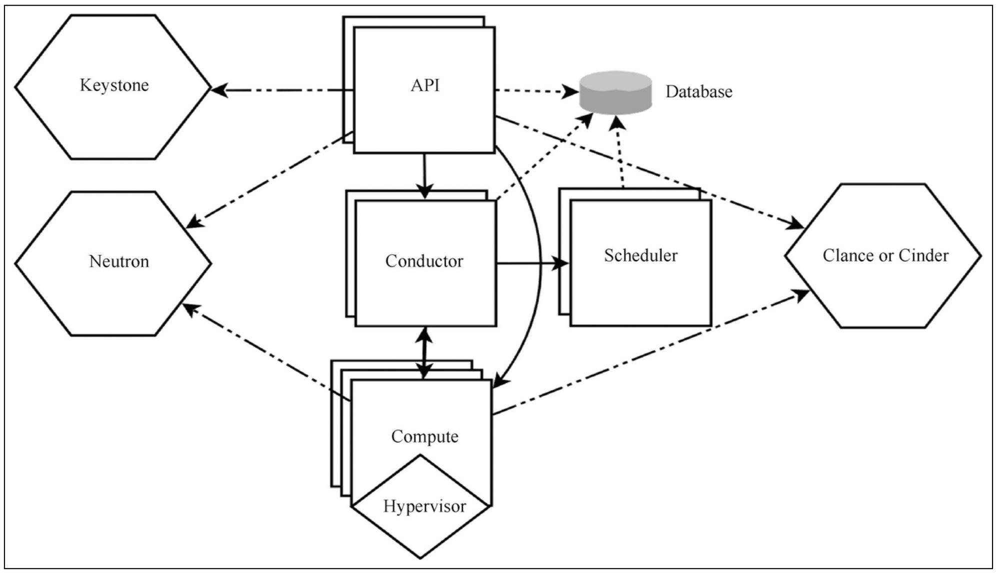


Figure 5.11: Nova architecture (Source[6])

5.5.2 Principles of OpenStack Computing Services

The Nova architecture consists of various components like Nova-API,Nova-scheduler,Nova-compute,Nova-conductor,Messager Queue. These components run as subservices (background Daemon processes), and their operating architecture is shown in **Figure 5.12**

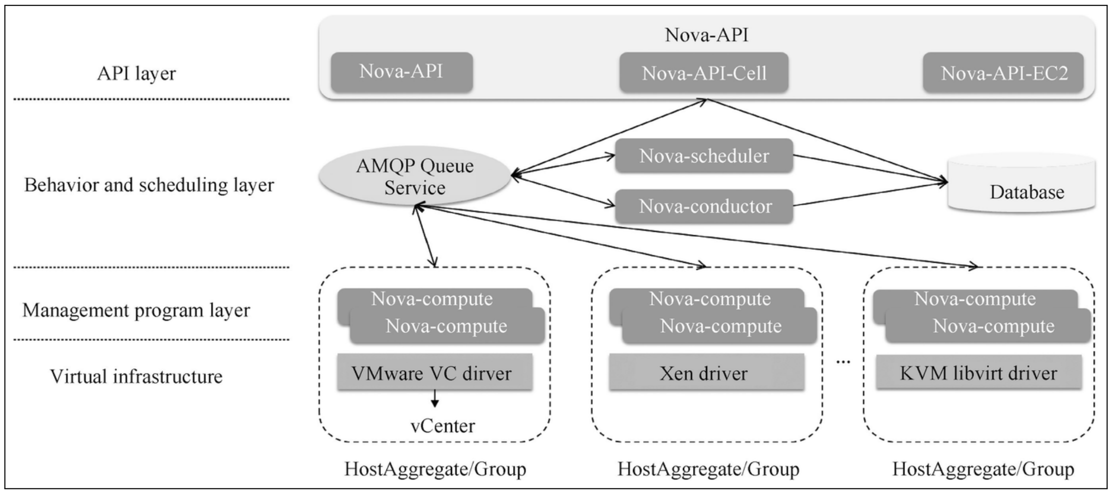


Figure 5.12: Nova operating architecture (Source[6])

- **Nova-API** serves as the entry point for the entire Nova component, receiving and responding to API calls from users. It exposes HTTP REST APIs that can be accessed by clients like

OpenStack CLI and Dashboard.

- **Nova-scheduler** is responsible for virtual machine scheduling, determining which compute node should run a particular instance based on resource requirements specified by the user. It uses filtering and weighting mechanisms to select the optimal compute node for instance creation.
- **Nova-compute** is the core service for managing virtual machines on compute nodes. It interacts with hypervisors, which are virtualization managers running on compute nodes, to handle the life cycle management of instances.
- **Nova-conductor** acts as an intermediary between Nova-compute and the database, handling database updates and queries on behalf of Nova-compute for enhanced security and scalability.
- **The Message Queue** is used for communication and coordination between the various Nova subservices. It decouples the subservices, allowing them to communicate through messages exchanged via the Message Queue.

Now, let's examine the workflow of Nova's subservices, which work together to create virtual machines. **Figure 5.13** shows the Nova service process.

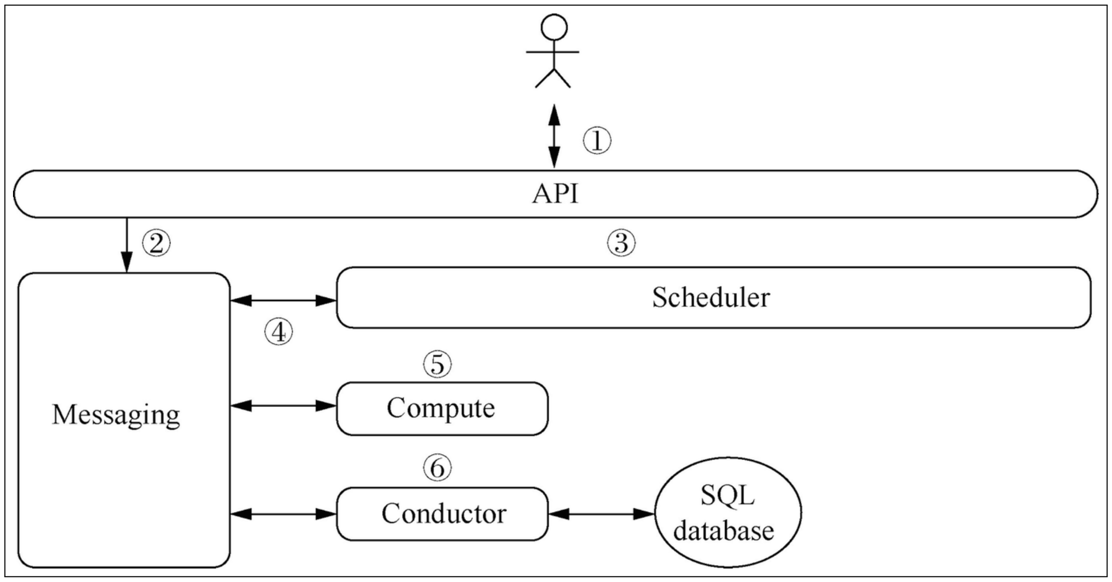


Figure 5.13: Nova Workflow (Source[6])

In the virtual machine creation process the following steps are followed,

1. The customer, who can be an OpenStack end-user or another program, sends a request to the Nova-API (OpenStack Compute API) to create a virtual machine.
2. The Nova-API receives the request and performs any necessary processing. Once the processing is complete, the Nova-API sends a message to a messaging system such as RabbitMQ, instructing the Scheduler to create a virtual machine.

3. The Scheduler, after receiving the message from the Nova-API via the messaging system, employs a scheduling algorithm to determine the most suitable compute node for hosting the virtual machine. It selects compute node A from the available compute nodes.
4. The Scheduler sends a message back to the messaging system, specifying that the virtual machine should be created on compute node A.
5. The selected Nova-compute node A receives the message from the Scheduler via the messaging system. It then initiates the process of creating the virtual machine on a hypervisor associated with the compute node.
6. While creating the virtual machine, the Nova-compute node A may need to retrieve or update information from the database. In such cases, it sends a message to the Nova-conductor, another component of OpenStack, using the messaging system. The Nova-conductor is responsible for accessing the database and handling queries or updates related to the virtual machine.

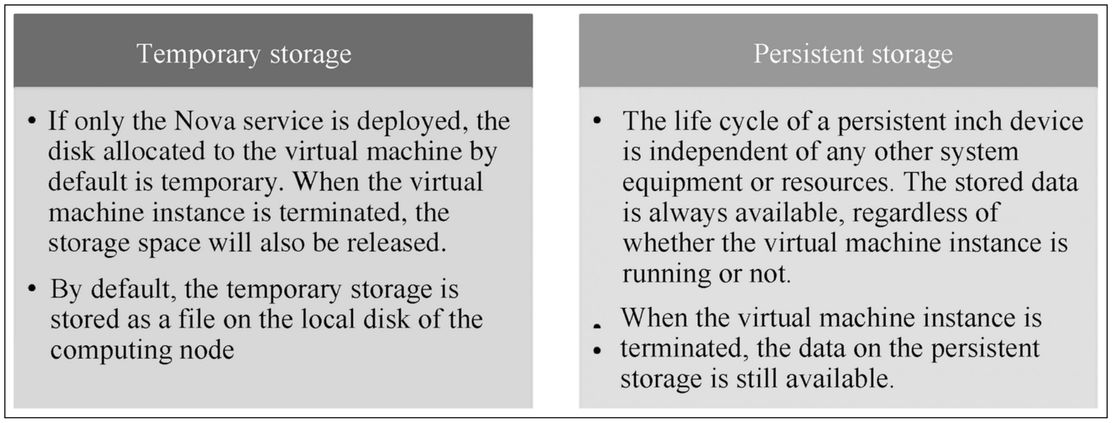
This sequence of steps demonstrates the flow of information and interactions among different components of OpenStack to fulfill the customer's request for creating a virtual machine. Understanding these steps helps us gain a deeper understanding of OpenStack as a whole.

5.6 OpenStack Storage Management

OpenStack provides a comprehensive set of services for managing storage resources in a cloud environment. From them users can choose freely based on their business needs. The storage management in OpenStack primarily revolves around two key components: Cinder(Block Storage) and Swift(Object Storage).

5.6.1 Introduction to OpenStack Storage Service

OpenStack's storage services play a crucial role as they are utilized by multiple service components. Storage is divided into Temporary Storage and Persistent Storage, as shown in **Fig. 5.14**.



Temporary storage	Persistent storage
• If only the Nova service is deployed, the disk allocated to the virtual machine by default is temporary. When the virtual machine instance is terminated, the storage space will also be released. • By default, the temporary storage is stored as a file on the local disk of the computing node	• The life cycle of a persistent inch device is independent of any other system equipment or resources. The stored data is always available, regardless of whether the virtual machine instance is running or not. • When the virtual machine instance is terminated, the data on the persistent storage is still available.

Figure 5.14: Storage classification in OpenStack (Source[6])

In OpenStack, virtual machines that utilize temporary storage losing all data upon shutdown, restart, or deletion. By deploying the Nova-compute service component, users can create such instances using the nova boot command, but these instances do not offer any security guarantees for the temporary storage.

Persistent storage in OpenStack can be categorized into three types: block storage, file system storage, and object storage. These storage types provide continuous data availability and security even when virtual machine instances are terminated. The hierarchy of these storage types is block storage, followed by file system storage, and finally object storage. File storage is typically built on top of block storage, while object storage often utilizes the local file system as its underlying or back-end storage mechanism.

- **Block storage** in OpenStack operates similarly to a directly attached hard disk, serving as direct storage space for the host and supporting database applications such as MySQL. There are two distinct types of block storage available:
  - DAS (Direct Attached Storage): In this configuration, each server has its dedicated storage that cannot be directly shared among multiple machines. To achieve sharing, operating system functions like shared folders are utilized.
  - SAN (Storage Area Network): SAN is a higher-cost storage method that involves the use of optical fiber and advanced equipment, providing exceptional reliability and performance. However, it should be noted that SAN equipment can be expensive, and the operational and maintenance costs associated with it are relatively high.
- **File system storage** differs from lower-level block storage. It operates at the application layer and typically refers to NAS (Network Attached Storage).NAS consists of network storage devices accessed through TCP/IP and commonly uses the Network File System (NFS) protocol. However, due to the involvement of the network and upper-layer protocols file system storage has a large overhead and the delay is higher than that of block storage. t is commonly used for data sharing among multiple cloud servers, such as centralized server log management and office file sharing.
- **Object storage** is a type of storage commonly used in self-developed applications (such as network disks). It combines the speed of block storage and the file sharing capability of file system storage. It is more intelligent and has its CPU, memory, network, and disk, which is higher level than block storage and file system storage. Cloud service providers generally offer REST APIs, which allow users to easily upload, download, and access files, simplifying integration with applications that utilize object storage services.

5.6.2 Principles of OpenStack Storage Service

Block Storage:

Block storage, also referred to as volume storage, allows users to access storage devices at the block level. Users can interact with block storage devices by mapping them to running virtual machine instances,enabling read/write operations, formatting, and other interactions. The block storage schema is shown in **Figure 5.15**

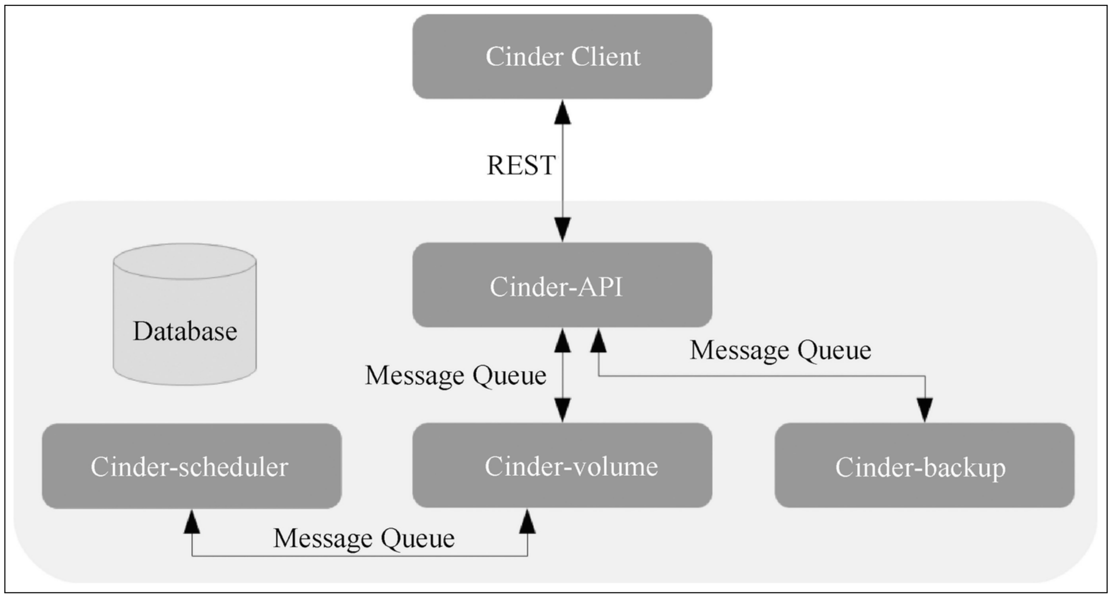


Figure 5.15: Block storage architecture (Source[6])

Block storage is persistent, meaning the data remains intact even when mapping changes between the storage device and virtual machine instances, or when the storage is remapped to a different virtual machine instance. In the OpenStack project, the Cinder component handles the provision of block storage. Cinder supports various back-end storage types, depending on the available storage options.

The Cinder component in OpenStack offers a block storage solution for virtual machine instances. It offers a wide range of functionalities for managing storage devices. These methods include volume snapshots, volume types and more. The type of block storage is determined by the drivers or back-end configurations, such as NAS, NFS, SAN, iSCSI, and Ceph. The Cinder-API and Cinder-scheduler services are typically deployed on control nodes, while Cinder-volume services can be deployed on control nodes, compute nodes, or standalone storage nodes.

The Cinder-API service acts as an interface for external communication, processing incoming requests such as create, delete, list, and show. On the other hand, the Cinder-scheduler service collects information about capacity and capabilities from the back-end systems, applies scheduling algorithms to assign volumes to specific Finder-volumes, and filters and weighs available back-end options. Multiple Cinder-volume services can be deployed across different nodes, each configured with specific settings to access different back-end devices. These services interact with the respective storage vendors' driver code to handle tasks such as collecting capacity information, managing volume operations, and maintaining device interactions. This distributed architecture allows for efficient and flexible management of block storage in OpenStack environments.

Figure 5.16 shows the process of creating a volume for Cinder, in which the Cinder components involved in OpenStack consist of the following service processes:

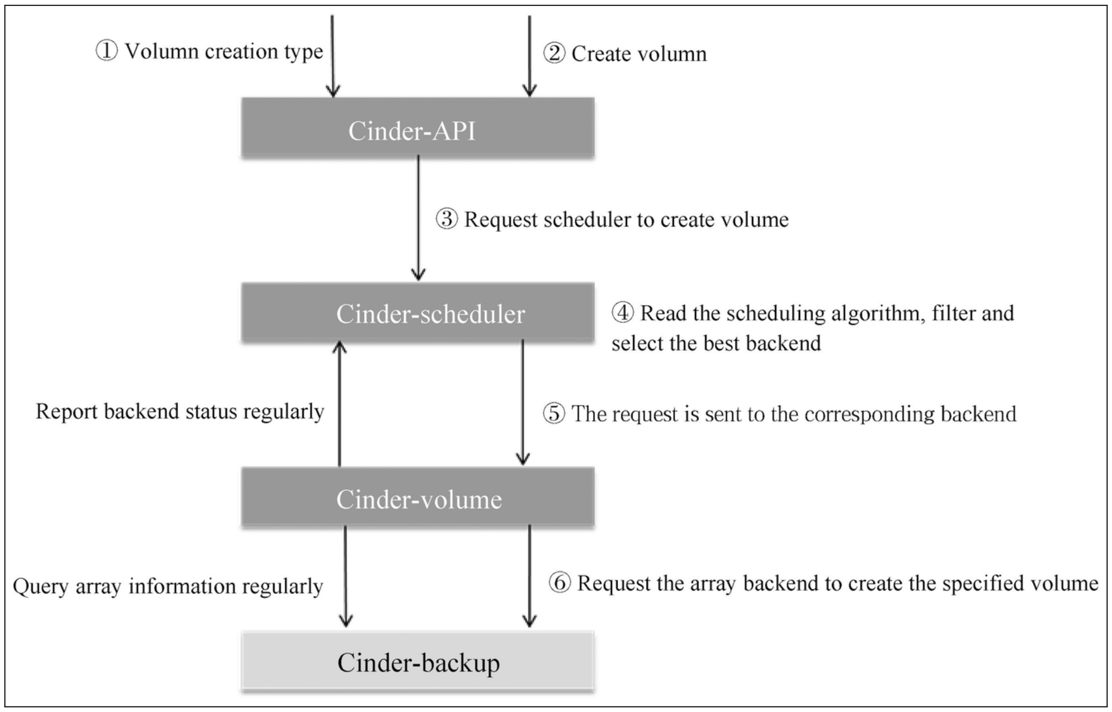

① Volumn creation type
② Create volumn
Cinder-API
③ Request scheduler to create volume
Cinder-scheduler
④ Read the scheduling algorithm, filter and select the best backend
Report backend status regularly
⑤ The request is sent to the corresponding backend
Cinder-volume
Query array information regularly
⑥ Request the array backend to create the specified volume
Cinder-backup

Figure 5.16: Cinder-volume creation process (Source[6])

- Cinder-API: Receives API requests and forwards them to Finder-volume.
- Cinder-volume: Interacts directly with block storage, handling tasks assigned by Cinder-scheduler and communicating through message queues. It also maintains the state of the block store and drives interactions with various types of storage.
- Cinder-scheduler: Selects the best storage node for volume creation, similar to Nova-scheduler.
- Cinder-backup: Provides backups for any type of volume.

Many storage drivers support virtual machine instances with direct access to the underlying storage, avoiding layer-by-layer transformations that consume performance and improving overall I/O performance. The Cinder components also support using common file systems as block devices. For example, in NFS and GlusterFS file systems, you can create a standalone file that maps to a virtual machine instance as a block device, similar to creating virtual machine instances in QEMU, where the files are saved in the "/var/lib/nova/instances" directory.

Object Storagr

The Swift component in the OpenStack project serves as a reliable and scalable solution for object data storage and retrieval. It provides a REST API for managing and accessing data stored as objects. Swift is typically deployed alongside the Keystone component to ensure proper authentication and authorization.

Key features of Swift include multi-tenancy support, cost-effectiveness, high scalability, and the ability to handle large amounts of unstructured data. It consists of several key sections:

- Proxy Server: Responsible for handling communication within the Swift architecture. It receives API and HTTP requests, manages objects, modifies metadata, and creates containers. The proxy server determines the location of accounts, containers, or objects using a ring and forwards requests accordingly. It also provides a graphical web interface for listing files or containers and utilizes MemCached for caching to enhance performance.
- Account Server: Manages accounts within the object storage system.
- Container Server: Manages containers and their associations with objects. The container server maintains a mapping between object stores and folders. It tracks statistics such as the total number of objects and container usage. Object information is stored in an SQLite database file, which is backed up in a clustered manner.
- Object Server: Stores, retrieves, and deletes actual object data. It is a binary large object storage server that utilizes a local device. Each object is stored using a path generated from a hash value of the object name and action timestamps. Deletions are treated as a new version of the file, indicated by a tombstone extension. This ensures proper handling of deleted files and prevents accidental data restoration.
- Periodic Processes: Handles regular transactions to ensure data continuity and effectiveness. This includes replication services, auditing services, update services, and deletion services.
- WSGI Middleware: Handles authentication-related tasks and establishes connections with Keystone components.
- Swift Client: Allows users with appropriate permissions to submit commands and perform actions from the client-side.
- Swift-Init: A script used for initializing the Ring file, which requires the daemon name as an argument and provides necessary action commands.
- Swift-Recon: A command-line interface (CLI) tool used for retrieving various performance metrics and status information for a Swift cluster.
- Swift-Ring-Builder: A tool used for creating and rebalancing the ring, which helps with the distribution of data across the cluster.

In Swift object storage, data is distributed evenly using software logic design. By default, three copies of the data are saved. he location where these three data copies are stored plays a crucial role in the overall performance of the cluster. Users have the option to save the copies on different hard drives within the same server or on different servers within the same rack. **Figure 5.17** Shows the swift data model. Swift object storage consists of three layers: Account, Container, and Object (account/container/object). These layers can be expanded without limitations in terms of the number of nodes per tier.

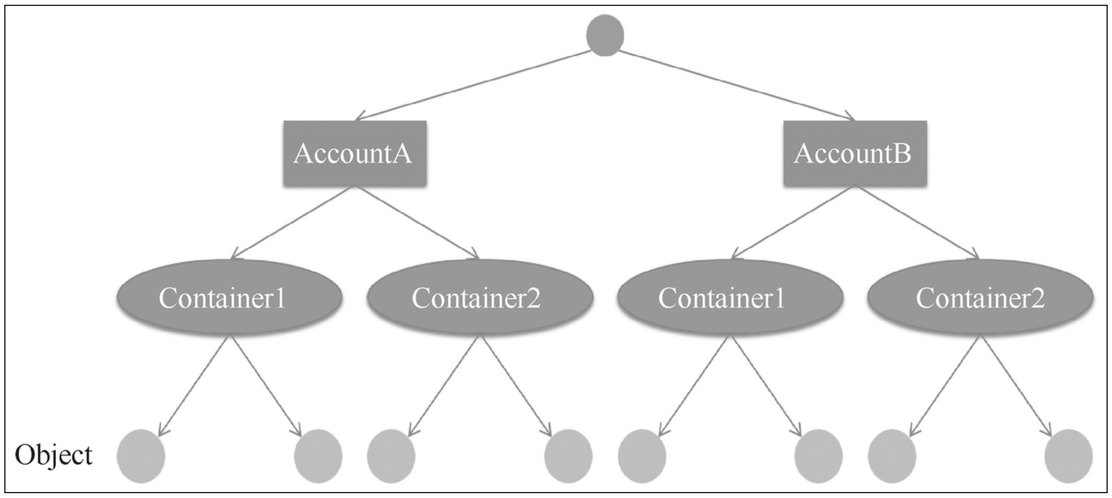


Figure 5.17: Swift Data Model (Source[6])

In Swift object storage clusters, when the host nodes storing data experience downtime, it puts a heavy load on the entire cluster. This is due to the need for data transfer and rebalancing, as multiple copies of the data are involved. To address this, various technologies such as network card aggregation and solid-state disks are utilized to enhance the overall performance of the cluster.

File system storage

Manila is the program code name for file system storage in the OpenStack project. It allows multiple users to mount and access a shared remote file system simultaneously. Manila supports various back-end storage drivers and offers functionalities like creating files, specifying access rules, taking snapshots for restoration, and monitoring usage. It provides a flexible and scalable solution for file system storage across multiple servers using different storage protocols.

5.7 OpenStack Network Management

OpenStack considers network as a crucial resource. Nova abstracts the virtual machine environment, while Swift and Cinder provide storage. However, without a network, virtual machines remain isolated. Initially, Nova-network module handled network services, but for enhanced topology, scalability, and support for multiple network types, Neutron was introduced as a dedicated component to replace Nova-network.

5.7.1 Basics of Linux Network Virtualization

Neutron abstracts and manages the two-tier physical network. Physical servers in a traditional network have multiple NICs connected to physical switches for communication between various applications. In order to communicate with each other, each physical server has one or more physical network cards (NICs) that are connected to physical switching devices, such as switches, as shown in **Figure 5.18**

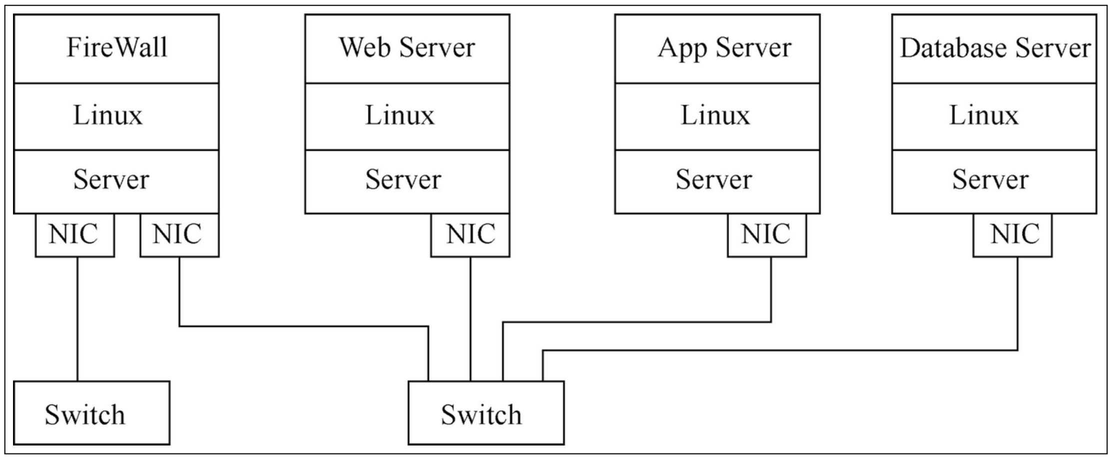


Figure 5.18: Traditional two layer switch (Source[6])

With virtualization technology, multiple operating systems and applications shown in **Figure 5.18** can share the same physical server as virtual machines managed by Hypervisor or VMM. The network structure shown in **Figure 5.18** has evolved into the virtual machine structure shown in **Figure 5.19**.Virtual machines use virtual network cards (vNICs) for network capabilities and Hypervisor can create one or more vNICs for each virtual machine. These vNICs are equivalent to NICs from the virtual machine's perspective. Virtual switches are used to achieve the network structure similar to traditional physical networks. Each vNIC is connected to a virtual switch, and these virtual switches connect to the external physical network through the physical server's NIC.

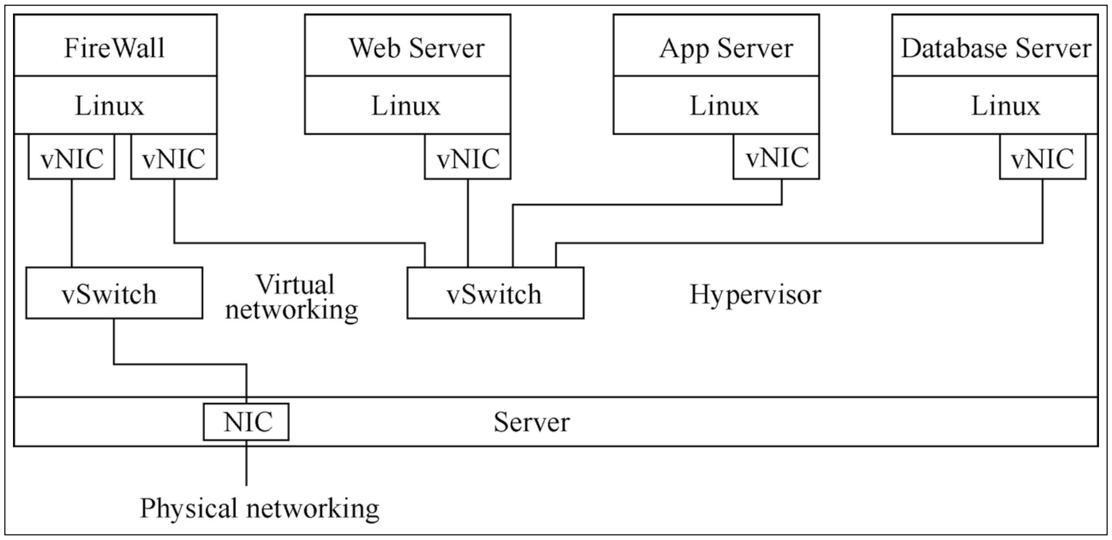


Figure 5.19: Virtual machine Network structure Evolve From Fig 5.18 (Source[6])

The virtualization of network devices in Linux environments involves various technologies, such as TAP, TUN, VETH devices, Linux Bridge, and OVS (Open vSwitch). TAP and TUN devices are virtual network devices implemented by the Linux kernel, allowing data to be sent between the kernel and user space programs. VETH devices come in pairs and redirect data between the ends. Linux Bridge acts as a virtual switch, binding network devices and virtualizing them as ports. OVS is a virtual switch that connects physical and virtual networks, allowing for network management, traffic monitoring, and configuration. It supports VLAN isolation, QoS configuration, and standard management interfaces like NetFlow and sFlow. OVS also supports OpenFlow, enabling management by an OpenFlow Controller. Distributed virtual switches, implemented by OVS, manage virtual switches across multiple hosts in a cloud environment. These technologies provide the necessary abstractions and functionalities to build and manage virtual networks effectively.

5.7.2 Introduction and Architecture of OpenStack Network Services

Neutron, unlike Nova and Swift, operates with a single major service process called Neutron-server. Running on a network control node, Neutron-server serves as the entry point for accessing Neutron[10] through its REST API. User HTTP requests received by Neutron-server are handled by various agents distributed across computing and network nodes. Neutron provides multiple API resources that represent different network abstractions, with the core resources being Network/-Subnet/Port for L2 abstraction. Other abstractions, such as routers and higher-level services, are considered extended resources. Neutron follows a plugin-based architecture to facilitate scalability, where each plugin supports a set of API resources and performs specific operations by calling the appropriate agent via RPC. Core Plugins, which support the underlying two-tier virtual network, must implement L2 abstractions, while service plugins, like firewall plugins, offer additional functionalities. For L3 abstraction, certain Core Plugins were not implemented, and in version H, Neutron introduced the L3 Router Service Plugin to provide router services. Agents, on the other hand, are responsible for practical tasks involving physical network devices or virtualization technologies, such as the L3 Agent for routing operations.

The Neutron architecture is shown in **Figure 5.20**

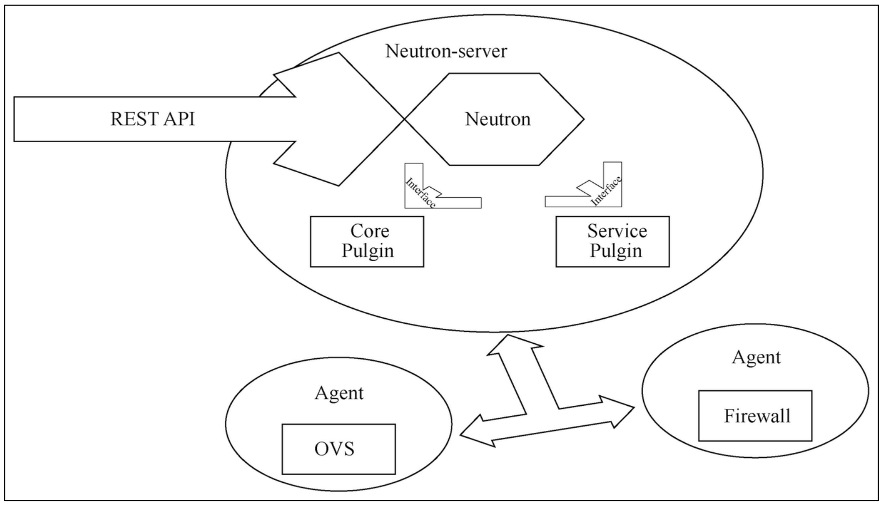


Figure 5.20: Neutron architecture (Source[6])

In version H, Neutron introduced the ML2 Core Plugin to address the issue of code duplication found in different Core Plugin implementations. The ML2 Core Plugin offers a more flexible structure that supports various Core Plugins in a Driver form. Its purpose is to replace the existing Core Plugins. The implementation of the ML2 Core Plugin and the various Service Plugins remains similar to what has been discussed in this chapter, even if Neutron is considered as a standalone project.

5.8 OpenStack Orchestration Management

The popularity of cloud computing has led to the emergence of multiple platforms. The success of a platform depends on its ability to effectively support complex application orchestration. OpenStack's Heat plays a crucial role in orchestration, contributing to OpenStack's leadership in cloud computing, specifically in Infrastructure as a Service (IaaS). This section explains OpenStack orchestration management, the concept of orchestration, Heat's role, Heat templates, implementation and support of Heat templates and Heat itself. It covers infrastructure management, software configuration and deployment, automatic resource scaling, load balancing, integration with configuration management tools, as well as integration with IBM UCDP/UCD.

5.8.1 Introduction to OpenStack Orchestration Service

Heat is a service that enables the orchestration of composite cloud applications using templates.It supports two template formats: Amazon's CloudFormation (CFN) template format and Heat's own HOT template format. Templates provide a simplified way to define and deploy complex infrastructure, services, and applications. The relationship between Heat and other modules is shown in **Figure 5.21**.

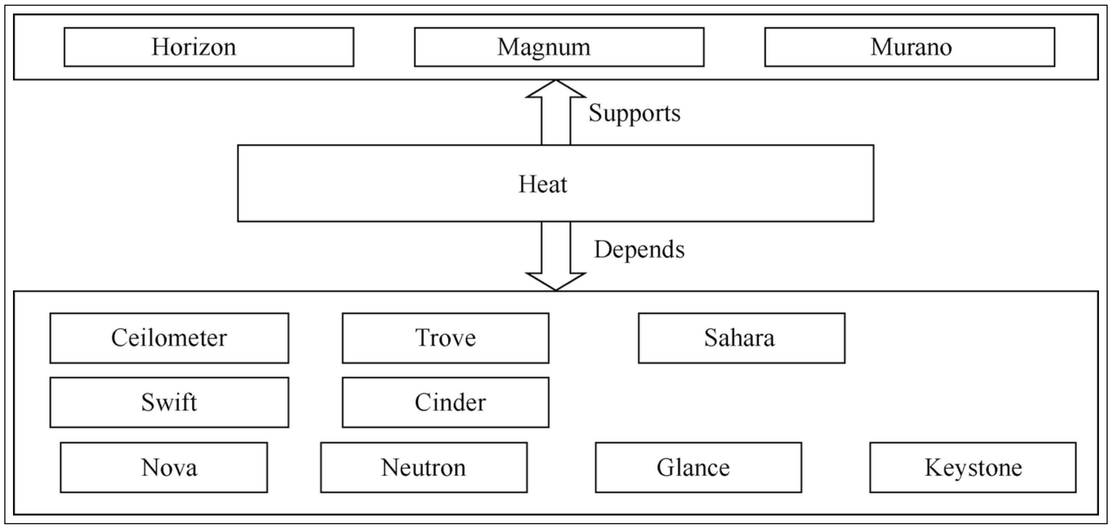


Figure 5.21: The relationship between Heat and other modules (Source[6])

Heat supports templates in two formats: JSON-based CFN templates and YAML-based HOT templates. CFN templates are designed to maintain compatibility with AWS, while HOT templates are specific to Heat and offer a wider range of resource types that align with Heat's capabilities.

A typical HOT template consists of the following elements:

- Template version: This is a required field that specifies the version of the template. Heat verifies the template based on its version.
- List of parameters: This is an optional field that includes a list of input parameters for the template.
- List of resources: This is a required field that defines various resources to be included in the resulting stack. Dependencies can be specified between resources, such as creating a port and using that port to create virtual machines.
- Output list: This is an optional field that provides information exposed by the resulting stack. This information can be used by users or passed as input to other stacks.

5.8.2 OpenStack Orchestration Service Architecture

Heat contains the following important components:

1. Heat-API: It implements the REST API and handles API requests by sending them to Heat-Engine via AMQP.
2. Heat-API-CFN: This component offers an API compatible with AWS CloudFormation and forwards requests to Heat-Engine using AMQP.
3. Heat-Engine: It provides the core functionality of Heat, enabling collaboration and orchestration of resources.

The Heat architecture is shown in **Figure 5.22**

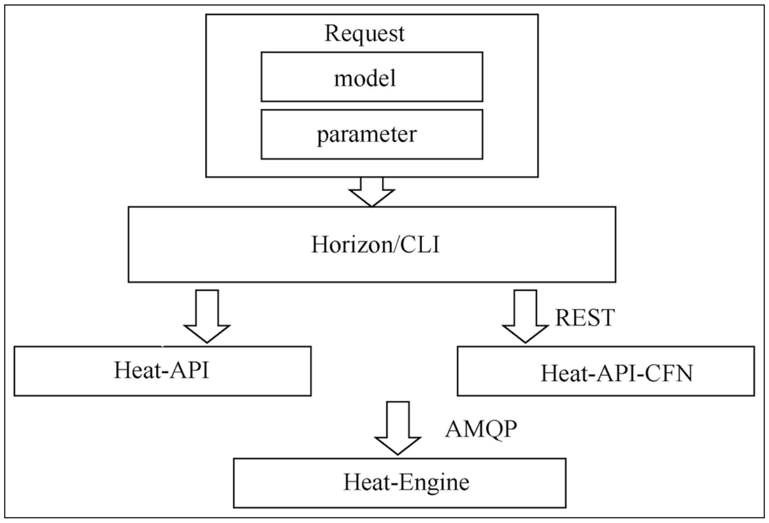


Figure 5.22: Heat architecture (Source[6])

When a user submits a request with templates and parameters through Horizon or CLI, it is converted into a REST API call and sent to either Heat-API or Heat-API-CFN. These components verify the correctness of the template and pass the request asynchronously to Heat-Engine using AMQP. Heat-Engine then resolves the request into various resources, interacting with other OpenStack service clients by sending REST requests. Through this parsing and collaboration process, the request is processed and finalized.

5.8.3 Principles of OpenStack Orchestration Service

OpenStack provides CLI and Horizon as management interfaces to handle resources for users. However, executing command lines and using a browser interface can be time-consuming and tedious. Writing and maintaining scripts for command-line operations and managing interdependencies can be challenging. Directly interacting with the REST API through programming introduces additional complexity and is difficult to maintain and extend. These limitations hinder the efficient management of OpenStack for large-scale operations and make it less suitable for resource orchestration to support IT applications.

In response to these challenges, Heat was developed. Heat utilizes popular industry templates to design and define orchestrations. Users can easily create orchestrations by writing templates in a text editor using key-value pairs. Heat provides several template examples to facilitate adoption. In most cases, users only need to choose the desired arrangement, copy and paste the relevant parts to complete the template.

Heat supports four types of orchestrations:

1. Heat's infrastructure orchestration: OpenStack provides infrastructure resources such as computing, networking, and storage. By orchestrating these resources, users can obtain the necessary virtual machines. It is worth mentioning that users can also provide simple scripts to configure virtual machines during the orchestration process.

2. Heat's software configuration and deployment orchestration: Users can configure virtual machines with complex setups, including software installation and configuration. Heat provides features such as Throughware Configuration and Software Deployment to facilitate software management.

3. Heat's automatic scaling of resources orchestration: For advanced requirements, such as automatically scaling a group of virtual machines based on load or load-balancing virtual machines, Heat offers support through features like AutoScaling and LoadBalance. Heat provides a wide variety of templates for reference when managing complex applications with scaling and load balancing needs.

4. Heat's load balancing orchestration: In complex applications or scenarios where users already have deployments based on popular configuration management tools like Chef, Heat allows the reuse of existing configurations, saving significant development or migration time.

In summary, Heat simplifies resource management and provides robust support for orchestrating IT applications within the OpenStack environment.

Chapter 6

Experiments

6.1 OpenStack

6.1.1 Installation

To install OpenStack on ubuntu 20.04 we use DevStack. DevStack installs all the components in one environment. This method assuming that a virtualization system is already installed. Here, we use Oracle VirtualBox 6.1[1].

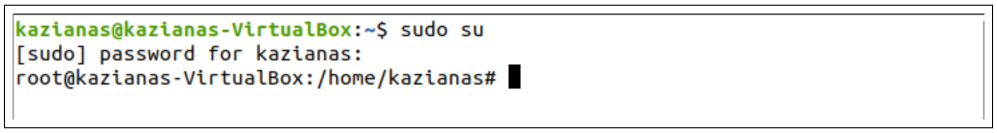
We cannot use the DevStack for production purposes because sometimes, after reboot, it is almost impossible to bring it up on the last state.

Follow these step:

1. Download Ubuntu 20.04 ISO.
2. Create Ubuntu 20.04 VM in Virtual Box
3. After successful installation of Ubuntu we can start OpenStack installation. For this we use following step in terminal.
4. In terminal first we switch to as a root user for full administrative privileges. For this first we run the following command:

   ```
           sudo su
   ```

   This command require password.After putting the password now we allow as a root user

   ```
   kazianas@kazianas-VirtualBox:~$ sudo su
   [sudo] password for kazianas:
   root@kazianas-VirtualBox:/home/kazianas#
   ```

[1]https://www.virtualbox.org/wiki/Download_Old_Builds_6_1

5. Before we start installing, we need to ensure that our system is updated, for that run following command:

```
        apt update -y && apt upgrade -y
```

6. After completing the upgrade ,the system required a reboot.To reboot the system we have to run a command:

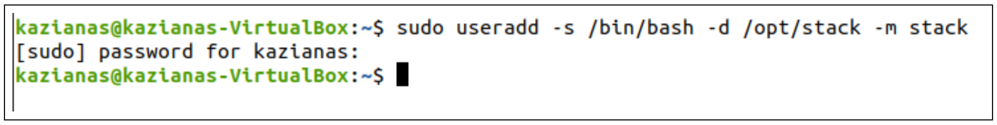
```
        sudo reboot
```

7. Now the important steps to install OpenStack on Ubuntu[14]. We create a new user named stack for our system to setup OpenStack, as it should be installed on a non-root user with sudo enabled. Open fresh terminal and run the following command:

```
        sudo useradd -s /bin/bash -d /opt/stack -m stack
```

   create a new user named "stack" with a specific shell, home directory, and the option to create the home directory if it does not already exist.

   Here the home directory is /opt/stack .The home directory is the default location where the user will be placed after logging in. Output :-

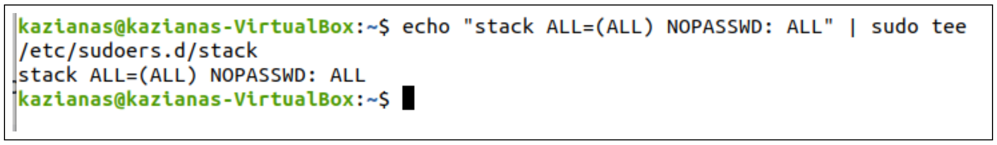


```
kazianas@kazianas-VirtualBox:~$ sudo useradd -s /bin/bash -d /opt/stack -m stack
[sudo] password for kazianas:
kazianas@kazianas-VirtualBox:~$
```

8. Now we also need to enable stack user to have root privileges and run without a password, for that we run the following command:

```
        echo "stack ALL=(ALL) NOPASSWD: ALL" | sudo tee /etc/sudoers.d/stack
```

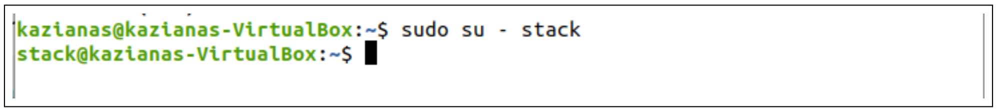
   The output will look like this:



```
kazianas@kazianas-VirtualBox:~$ echo "stack ALL=(ALL) NOPASSWD: ALL" | sudo tee
/etc/sudoers.d/stack
stack ALL=(ALL) NOPASSWD: ALL
kazianas@kazianas-VirtualBox:~$
```

9. Now we log in to the stack user which we create above. For this we run the following command:

```
    sudo su - stack
```



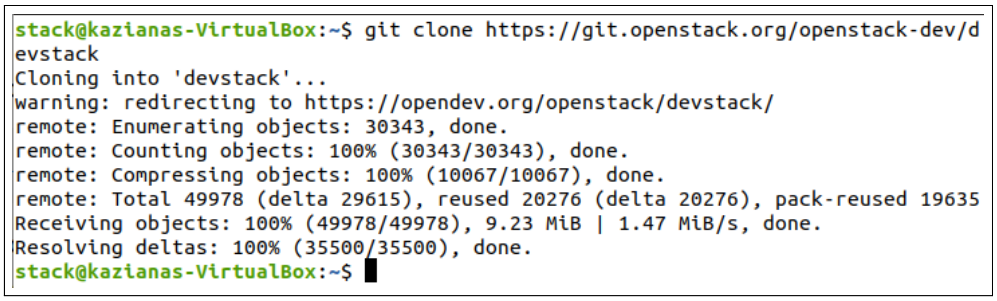
```
kazianas@kazianas-VirtualBox:~$ sudo su - stack
stack@kazianas-VirtualBox:~$
```

10. If git is not installed in my system ,to download git we enter this command

```
sudo apt install git -y
```

11. Now we download DevStack. For this enter this command to download/clone DevStack from its repository to our system

```
git clone https://git.openstack.org/openstack-dev/devstack
```

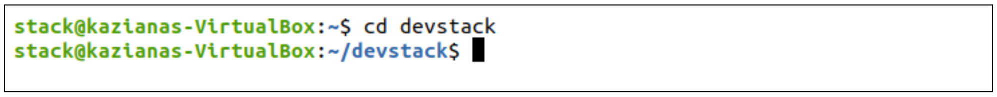


```
stack@kazianas-VirtualBox:~$ git clone https://git.openstack.org/openstack-dev/d
evstack
Cloning into 'devstack'...
warning: redirecting to https://opendev.org/openstack/devstack/
remote: Enumerating objects: 30343, done.
remote: Counting objects: 100% (30343/30343), done.
remote: Compressing objects: 100% (10067/10067), done.
remote: Total 49978 (delta 29615), reused 20276 (delta 20276), pack-reused 19635
Receiving objects: 100% (49978/49978), 9.23 MiB | 1.47 MiB/s, done.
Resolving deltas: 100% (35500/35500), done.
stack@kazianas-VirtualBox:~$
```

DevStack repo contains a script stack.sh, which we will use to setup OpenStack. It also contains templates for configuration files.

12. Now we creating configuration(.conf) file for DevStack. For this first we go to the directiry called devstack by run the command

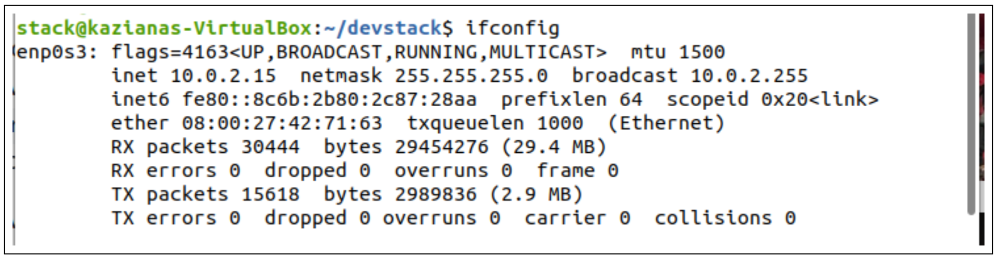
```
cd devstack
```



```
stack@kazianas-VirtualBox:~$ cd devstack
stack@kazianas-VirtualBox:~/devstack$
```

13. Then we check the IP address of our VM. For this we run

```
ifconfig
```

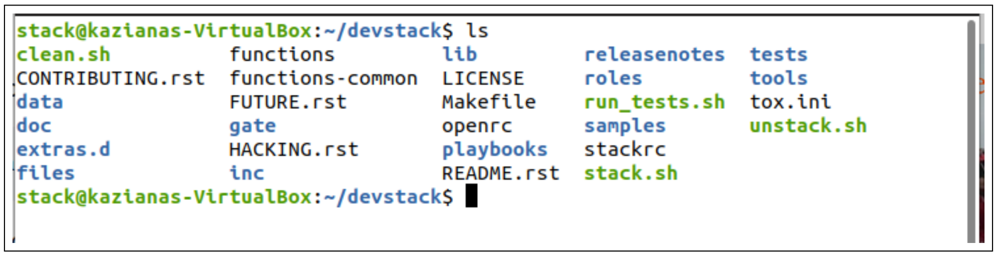


```
stack@kazianas-VirtualBox:~/devstack$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
        inet6 fe80::8c6b:2b80:2c87:28aa  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:42:71:63  txqueuelen 1000  (Ethernet)
        RX packets 30444  bytes 29454276 (29.4 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 15618  bytes 2989836 (2.9 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

14. Afterward, we Create a **local.conf** file by copying the sample configuration file. For this we run

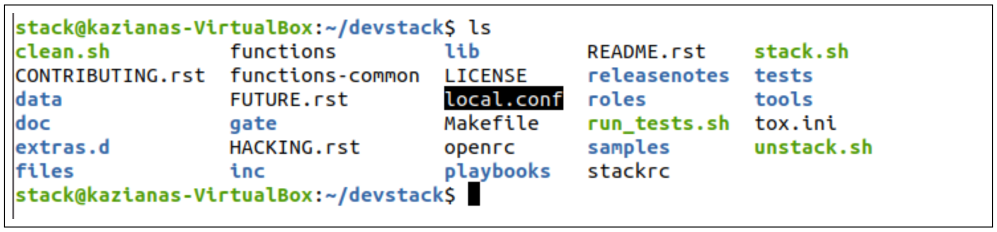
```
cp samples/local.conf local.conf
```

The command "cp samples/local.conf local.conf" is used to create a copy of the file "local.conf" from the "samples" directory and save it as "local.conf" in the current directory.



```
stack@kazianas-VirtualBox:~/devstack$ ls
clean.sh          functions         lib          releasenotes  tests
CONTRIBUTING.rst  functions-common  LICENSE      roles         tools
data              FUTURE.rst        Makefile     run_tests.sh  tox.ini
doc               gate              openrc       samples       unstack.sh
extras.d          HACKING.rst       playbooks    stackrc
files             inc               README.rst   stack.sh
stack@kazianas-VirtualBox:~/devstack$
```

Figure 6.1: Before Creating Local



```
stack@kazianas-VirtualBox:~/devstack$ ls
clean.sh          functions         lib          README.rst    stack.sh
CONTRIBUTING.rst  functions-common  LICENSE      releasenotes  tests
data              FUTURE.rst        local.conf   roles         tools
doc               gate              Makefile     run_tests.sh  tox.ini
extras.d          HACKING.rst       openrc       samples       unstack.sh
files             inc               playbooks    stackrc
stack@kazianas-VirtualBox:~/devstack$
```

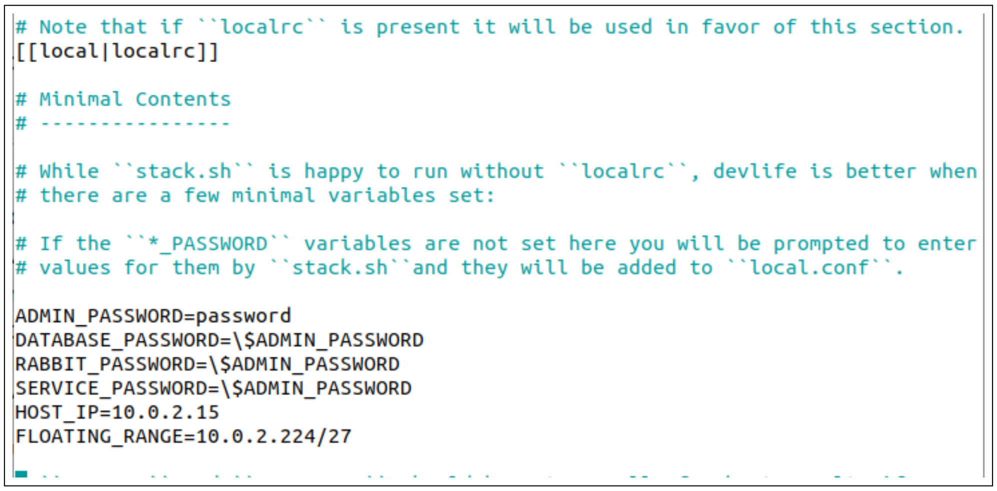
Figure 6.2: After Creating Local

15. we edit the configuration file by using nano. For this first we run:

```
nano local.conf
```

Then we paste this

```
[[local|localrc]]
ADMIN_PASSWORD=password
DATABASE_PASSWORD=$ADMIN_PASSWORD
RABBIT_PASSWORD=$ADMIN_PASSWORD
SERVICE_PASSWORD=$ADMIN_PASSWORD
HOST_IP=10.0.2.15
```



```
# Note that if ``localrc`` is present it will be used in favor of this section.
[[local|localrc]]

# Minimal Contents
# ----------------

# While ``stack.sh`` is happy to run without ``localrc``, devlife is better when
# there are a few minimal variables set:

# If the ``*_PASSWORD`` variables are not set here you will be prompted to enter
# values for them by ``stack.sh``and they will be added to ``local.conf``.

ADMIN_PASSWORD=password
DATABASE_PASSWORD=\$ADMIN_PASSWORD
RABBIT_PASSWORD=\$ADMIN_PASSWORD
SERVICE_PASSWORD=\$ADMIN_PASSWORD
HOST_IP=10.0.2.15
FLOATING_RANGE=10.0.2.224/27
```

16. Once the DevStack is installed and local.conf is setup properly as per our environment, then run the following command to start the installation process:

```
./stack.sh
```

The installation process may take 10 to 20 minute to complete, depending on your internet speed and system resources.

Above script will install keystone, glance, nova, placement, cinder, neutron, and horizon.

At the very end of the installation, you will get the host's IP address, URL for managing it and the username and password to handle the administrative task.

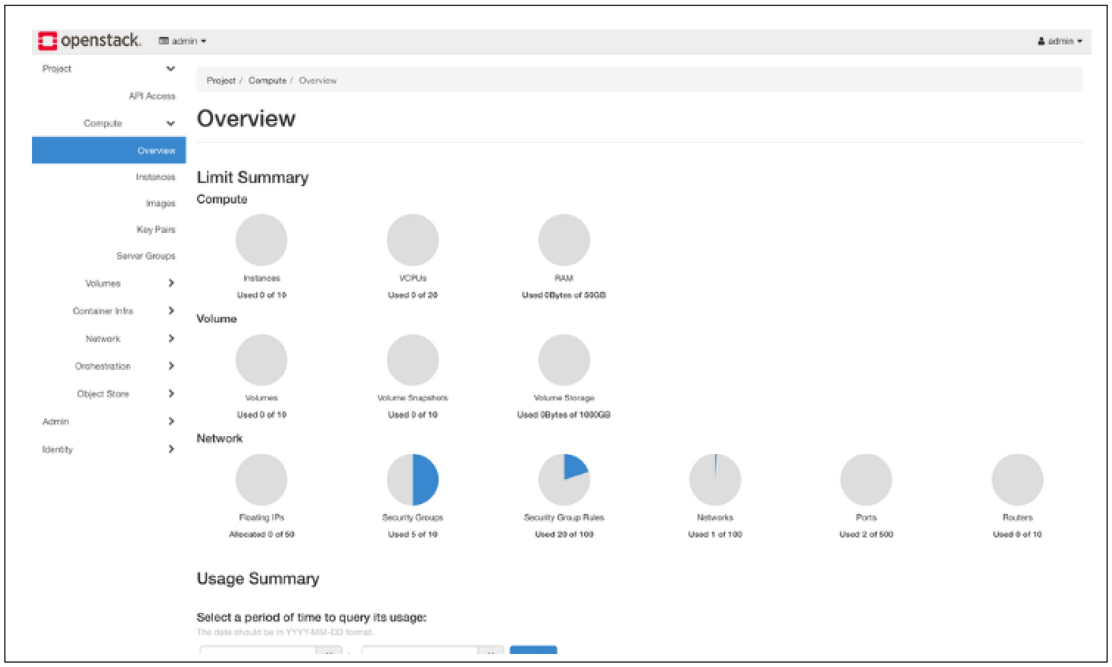
6.2 Using OpenStack Cloud Services

After Successfully installed OpenStack we can use it in different ways.

6.2.1 GUI

Login To Dashboard on a browser

Horizon is the name of the default OpenStack dashboard, which provides a web based user interface to OpenStack services. It allows a user to manage the cloud.Copy the horizon URL given in the installation output and paste it into your browser.To login to OpenStack with the default username - admin or demo and configured password - password(which I set when edit local.conf). Once you login into the OpenStack, you will be redirected to the Dashboard of OpenStack.



Create a Project in OpenStack Horizon

In OpenStack, projects are used to partition the cloud and users are associated with projects with different access levels defined by roles. Administrators can set resource limits for each project by adjusting quotas.

Now we'll cover creating a project, adding users, adjusting project quotas, and note that the Horizon interface remains consistent across OpenStack deployments.

There are three root-level tabs on the left menu in Horizon: Project, Admin, and Identity. Only users with administrative privileges can see the admin tab.

To create your first project,

1. **Navigate to Identity→Projects.**
2. **Click the Create Project button near the top right to create a new project.**
3. **Specify a name for the project in the Name field, such as "Test" and click Create Project.**

You can also add Project Members and Project Groups.The created project will be listed on the Project Listing page. From there, as the admin user, you can view and adjust quotas, which are limits on resources like instances.

Create a User and Associate with Project

To create a new user and login with the admin user,

1. **Navigate to Identity→Users.**

   There are default users listed during cloud deployment should not be modified.

2. **Click the Create User button.**

3. Then we set values for **User Name, Password, Primary Project, and Role.** Email field is optional but helpful for password resets. Choose the project we created earlier from the Project dropdown. Role options include reader, member, admin, and more. Reader has the least authority.

4. **Press Create User to create the user.**

5. Now to log in this user log out of Horizon as admin, and log back in with our new user. After log in we see current project is displayed top left, user is shown top right in Horizon.

Managing and Creating Images

We will learn how to upload images to Glance through Horizon and how to create an image from an instance snapshot.Glance uses Ceph to store images instead of the local file system.

To upload an image,

1. **Navigate to Project→Compute→Images.**

   This tab contains a list of all your images within OpenStack.

2. **Click the Create Image button.**

   Choose QCOW2 – QEMU emulator as the recommended image format during creation. Ensure you have the image locally before uploading it through Horizon.

3. **Then click the Create Image button near the down right.**

Create an Instance

To create an instance, including setting up a private network and router, creating a security group, and how to add an SSH key pair.

- **Create a Private Network:** To create a private network,
  1. **Navigate to Project→Network→Networks.**
  2. Then **Click Create Network.**
  3. In **Network** tab we fill following details:
     (a) **Network Name:** Set a name for the network. This example is called Private.
     (b) **Enable Admin State:** Leave this checked to enable the network.
     (c) **Create Subnet:** Leave this checked to create a subnet.
     (d) **Availability Zone Hints:** Leave this option as default.
  4. Next move to the **Subnet** tab and fill following details:
     (a) **Subnet Name:** Set a name for the subnet. This example subnet is called private-subnet.

    (b) **Network Address:** Select a private network range.
    (c) **IP Version:** Leave this as IPv4.
    (d) **Gateway IP:** This is optional. If unset, a gateway IP is selected automatically.
    (e) **Disable Gateway:** Leave this unchecked.

  5. For now we will keep the default details in the **Subnet Details** tab.In **Subnet Details** you are able to turn on DHCP server, assign DNS servers to your network and set up basic routing.
  6. **Click Create(near the down right) to create the network**

- **Create a Router:** You next need to create a router to bridge the connection between the private network and the public network. The public network is called External.To create a router,

  1. **Navigate to Project→Network→Routers.**
  2. **Click Create Router.**
  3. We fill following details as below:
     (a) **Router Name:** Set a name for the router here. This example router is called Router.
     (b) **Enable Admin State:** Leave this checked to enable the router.
     (c) **External Network:** Choose the network External.
     (d) **Availability Zone Hints:** Leave this as the default.
  4. After filling detail click on **Create Router.**

- **Connect Router to Private Network:** Next connect the router to the private network by attaching an interface to enable network communication between the private and external networks.

  To connect the router to private network,

  1. **Navigate to Project→ Network→Routers.**
  2. Click the name of the router to access its details page. There are three tabs: Overview, Interfaces, and Static Routes.
  3. To attach an interface,
     (a) **navigate to the Interfaces tab.**
     (b) Then click **Add Interface** near the top right.
     (c) We fill following details:
         i. **Subnet:** Choose the private-subnet,which is previously created.
         ii. **IP Address:** Fill in IP Address. If we don't set an IP address one is selected automatically.
     (d) Click **Submit** to attach the Private network to this router.
  4. The interface is then attached. To view this **navigate to Project→ Network→ Network Topology.**

- **Security Groups:** Security groups manage network traffic to and from instances. For example, port22 can be opened for SSH for a single IP or a range of IPs.

  To create a security group for SSH access,

  1. **Navigate to Project→ Network→ Security Groups.**
  2. **Click Create Security Group near the top right.**
  3. Then we fill following details:
     (a) **Name:** Name the group as we wish.
     (b) **Description:** Give if any.
  4. Then click **Create Security Group.**
  5. After creating the security group, add a rule to allow SSH traffic. Allow SSH traffic from the first hardware node in the cloud to this instance. To add a rule, load the form by navigating to Add Rule near the top right. We'll need to obtain the IP address of the first hardware node of your cloud. In the Add Rule menu, add the following information:
     (a) **Rule:** Select a secruity group. When adding rules you can choose from predefined options.
     (b) **Description:** Optional. Provide a description of the rule.
     (c) **Remote:** Select CIDR.
     (d) **CIDR:** Specify the IP address of your first hardware node.

We now have almost everything in place to create an instance.To create an instance,

1. **Navigate to Project→Compute→ Instances.**
2. Then click the **Launch Instance** button.
3. On the **Details** tab fill the following details:
   (a) **Instance Name:** Set a name for the instance.
   (b) **Description:** Optional. Set a description if this applies.
   (c) **Availability Zone:** Leave as the default, which is nova.
   (d) **Count:** Controls the number of instances spawned. Just create 1.
4. Next, move to the **Source** tab fill the following details:
   (a) **Select Boot Source:** In this example, we use Image as the boot source.
   (b) **Create New Volume:** Leave this checked as Yes. This creates a new Cinder volume where the specified operating system image is copied into it. The volume ultimately exists with the Ceph cluster, in the vms pool.
   (c) **Volume Size:** Allow the system to determine this for you.
   (d) **Delete Volume on Instance Delete:** Leave this option set as No. If checked, when the instance is deleted, the volume is as well.
   (e) Under the **Available** section, select the appropriate operating system. Clicking the up arrow will move it to the **Allocated** section.

5. Next, move to the **Flavor** tab. Flavors define the VCPUs, RAM, and disk space of an instance. Choose a suitable flavor from the available options.**Click the up arrow to move it to the Allocated section.**

6. Next, move to the Networks tab.In this section, choose the Private network associated with the instance. It's generally recommended to avoid selecting the External network unless you specifically require internet connectivity and use a floating IP instead. By default, if no private network is created, instances in a default cloud are associated with the External network, which means they consume a public IP and can be accessed over the internet.

7. Next, skip over the **Network Ports** tab and move to the **Security Groups.** This is where you select security groups for the instance.Then select a security group from **Available** section by click the up arrow. Then security group shows in **Allocated** section.

8. As the final step, move to the **Key Pair** tab. Here we specify an SSH public key to inject into the instance. We can upload your key at this stage using this form using the Import Key Pair button. We can also create a key pair on this tab.

   We will create a key pair from the first hardware node in our cloud so this instance will be accessible over SSH from that node.

   To create the SSH key pair from the first hardware node,

   (a) Login to the first hardware node.For this we run command

   ```
           sudo ssh -i ~/root/your_key_name root@hardware_node_ip
   ```

   (b) After logging in to the node, use

   ```
           ssh-keygen
   ```

   For generate an SSH key pair.The private key is saved in the default location of /root/.ssh/id_rsa.pub and a passphrase is set for additional security.To view the contents of the public key, use

   ```
           cat /root/.ssh/id_rsa.pub
   ```

   (c) Copy the entire key. It starts with "ssh-rsa".

   (d) Now back to the **Key Pair** tab. Click **Import Key Pair.** And then fill the following:

      i. **Key Pair Name:** Set a name for the SSH public key. It can be anything you like.
      ii. **Key Type:** This example uses an SSH Key key type.
      iii. **Public Key:** Paste in the public key you just copied.

   (e) Click **Import Key Pair.**

9. Afterthat, We click on **Launch** button.When build process is complete, the instance appears in the Instances Listing page.

Assign and Attach Floating IP

The previously created instance is connected to a private network, accessible only from within the cloud's hardware nodes. To access the instance externally, you can utilize a floating IP as an alternative method.

To allocate a floating IP,

1. First navigate to Project→Network→Floating IPs.
2. Then click **Allocate IP to Project.**
3. In the popup, make sure Pool is set to External (and optionally add a description)
4. Then click **Allocate IP** to add this floating IP address for use. In the same section, allocate the IP to a instance(which I create previously) by clicking the **Associate** button at the far right. Fill the following details:
   (a) **IP Address:** This field comes pre-selected with the floating IP, so there's no need to change anything here.
   (b) **Port to be associated:** Select the instance created previously.
5. Then click **Associate.** This instance is now accessible over SSH from the first hardware node of your cloud.

To login to this instance, after you login to your hardware node, run the following command [you will have to change the IP address to the one you just associated] :

```
    ssh -i /root/.ssh/id_rsa os_name@floating_ip_previously_create
```

6.2.2 Command Line

So far we've been learning how to manage OpenStack through a web browser. But it is also possible to manage through the command line using OpenStack's CLI called OpenStackClient.

Using the command line to manage your cloud introduces more flexibility in automation tasks and can generally make an administrator's life simpler. Let's learn how. We'll now install the OpenStackClient on the instance we just created above.

Before installing OpenStackClient, you must obtain two files from Horizon, which are required to prepare your shell environment. Those two files are clouds.yaml and the OpenStack RC file.

To install OpenStackClient,

1. First you need to acquire two essential files from Horizon: clouds.yaml and the OpenStack RC file. The clouds.yaml file serves as a configuration source for connecting to a cloud, while the OpenStack RC file provides authentication details for your user and project, enabling access to OpenStack resources.To collect these files,
   (a) **Log in to Horizon as your user.**
   (b) **Navigate to Project→API Access.**

(c) Then click Download **OpenStack RC File.**

(d) Download the OpenStack clouds.yaml and the OpenStack RC files to our machine.

2. Use SSH to log in to the instance created previously.

3. Now we prepare clouds.yaml and OpenStack RC files previously download.

   (a) Prepare the previously obtained clouds.yaml file for this instance. Save it as `~/.config/openstack/clouds.ya` copying the downloaded contents from Horizon to your machine. For this we run the following command

   ```
           mkdir -p ~/.config/openstack
   ```

   Then

   ```
           vi ~/.config/openstack/clouds.yaml
   ```

   To transfer the contents of the local clouds.yaml file to the instance, open the local file in a text editor, copy all the text, and paste it into the newly created version of clouds.yaml on the instance.

   After you paste the text of the file into the vi editor in you terminal, use the command `:wq` to save and quit the editor.

   (b) Next copy the OpenStack RC file contents from your local machine to the instance. You can place the file anywhere, but for this example, let's store it in the user's home directory. The full path will be `~/Development-openrc.sh.` Create and start editing the file with the following command

   ```
           vi ~/Development-openrc.sh
   ```

   Similar to before, open the local version of the file, copy all the text, and paste it into the instance's file open in your terminal. Save and exit the editor using the command `:wq`.

   (c) Now you have to run the file. First change the permissions:

   ```
           chmod +x Development-openrc.sh
   ```

   (d) Then, run the file:

   ```
           ./Development-openrc.sh
   ```

   You will have to enter your OpenStack password.

   (e) Then run the command:

   ```
           source Development-openrc.sh
   ```

4. Now we create a Python virtual environment so we don't interfere with the system's Python version.This is depend on OS. For example in CentOS 8 Stream, the system's Python executable is `/usr/libexec/platform-python` and is what will be used to create the virtual environment. Run below command

```
/usr/libexec/platform-python -m venv ~/venv
```

to create a virtual environment in path `~/venv`.

5. Now we active the Python virtual environment. For this we run the following command

```
source ~/venv/bin/activate
```

   After you have activated the virtual environment, the name of it will appear at the beginning of the command line.

6. Before installing OpenStackClient and to aid in a smooth installation we have to upgrade the pip by run below command

```
pip install --upgrade pip
```

7. With everything prepared, OpenStackClient can be installed. There exist two OpenStack-Client packages: python-openstackclient and openstackclient. Here we install python-openstackclient by run below command

```
pip install python-openstackclient
```

8. For an initial command, list the servers associated with your project by running

```
openstack server list
```

   Here, we can see the server created previously.

Command Structure

When working with OpenStackClient, the common command pattern starts with "openstack." To execute OpenStack commands, you can use "openstack" as a standalone command to enter the shell. Once in the shell, you no longer need to prefix commands with "openstack."

- **List all Available Subcommands:** Use `openstack --help` to list all available subcommands. You initially see all the flags you can pass, but after scrolling a bit, the subcommand list starts

- **Learn more about a Subcommand:** After seeing available commands, learn more about a command by using `openstack help <command>.`

- **List Items and Show Details:** It is very common when using OpenStackClient to list items and the command form is typically `openstack <subcommand> list.` This command lists all servers for the currently configured project.

  Furthermore, more information about an item can be found by typically running `openstack <subcommand> show <item>.` This command shows the details.

6.3 Python OpenStack SDK

The Python OpenStack SDK[7] (Software Development Kit) is a set of Python libraries that provide a high-level interface for interacting with OpenStack services. It simplifies the process of authenticating, making API calls, and managing resources within an OpenStack deployment.It offers a unified interface for working with various OpenStack services, abstracting the differences between different service APIs and providing a consistent experience.

Installing the SDK

The SDK package can easily be installed using Python's "pip" package manager.

```
pip install openstacksdk
```

To check the installed version you can call the module with:

```
python -m openstack version
```

Configuration and Connection

There are three ways to connect to Jetstream using the OpenStack SDK:

1. **Hard-coding:** You can directly include all the required connection parameters in the source code of your program.
2. **Environment variables:** If you run the application in an environment where the OS_* environment variables are defined, you can use them to automatically set up the connection.
3. **Configuration file:** Another option is to create a configuration file where you specify the name of the connection you want to use. The SDK will read this file and establish the connection accordingly.

The SDK developers recommend that you connect using a configuration file. Now create an authentication session by providing the required authentication details, such as the OpenStack identity URL, username, password, and project (tenant) name. You can authenticate using different methods, such as password authentication or token authentication. Here's an example of password-based authentication:

```
from openstack import connection

# Create a connection object
conn = connection.Connection(auth_url='https://<keystone-url>/v3',
                             username='<your-username>',
                             password='<your-password>',
                             project_name='<your-project-name>',
                             user_domain_name='<user-domain-name>',
                             project_domain_name='<project-domain-name>')
```

Once authenticated, you can use the connection object (conn) to interact with different OpenStack services. You can create, retrieve, update, and delete resources using the provided methods. For example, to list instances using Nova:

```
from openstack import compute

# List instances
for server in conn.compute.servers():
    print(server.name)
```

Similarly, you can use other service modules like openstack.network, openstack.object_store, etc., to work with specific services.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In conclusion, this thesis has made some contribution towards the automation of networkin using OpenStack. We explored the management of OpenStack using Horizon. We also discussed command-line management using OpenStackClient and the Python OpenStack SDK for automation and resource management. Through the step-by-step instructions provided, we demonstrated how to allocate and attach a floating IP to an instance, enabling external access to the instance from outside the cloud's hardware nodes. OpenStackClient, the command-line interface for OpenStack, which provides administrators with more flexibility in automation tasks. By installing and configuring OpenStackClient, we manage OpenStack resources efficiently and perform common operations such as listing instances.

7.2 Future Work

There are several promising areas for future research and development within the context of this thesis. One area of focus could be advanced networking and security. This involves exploring technologies like software-defined networking (SDN) and network function virtualization (NFV) to enable more dynamic network configurations and enhancing security mechanisms within OpenStack.

Automation and orchestration present another important direction for future work. Developing advanced automation frameworks can optimize the deployment and management of intricate cloud environments. This includes automating resource provisioning, workload scaling, and application life cycle management, ultimately improving operational efficiency.

We can focus on monitoring and analytics to provide real-time insights into the performance and usage of OpenStack resources.

Bibliography

[1] Central office re-architected as a datacenter. https://opennetworking.org/cord/. Accessed: 2023-05-20.

[2] Cloudify. https://docs.cloudify.co/index.html. Accessed: 2023-05-20.

[3] Mininet. http://mininet.org/. Accessed: 2023-05-20.

[4] Open network automation platform. https://www.onap.org/. Accessed: 2023-05-20.

[5] Open source mano. https://osm.etsi.org/. Accessed: 2023-05-20.

[6] Openstack. https://link.springer.com/chapter/10.1007/978-981-19-3026-3_6#Sec14. Accessed: 2023-05-29.

[7] Openstack python sdk. https://cvw.cac.cornell.edu/JetstreamAPI/openstacksdkworkflows. Accessed: 2023-05-29.

[8] Tacker - openstack nfv orchestration. https://wiki.openstack.org/wiki/Tacker. Accessed: 2023-05-20.

[9] Tosca version 2.0. https://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html. Accessed: 2023-05-20.

[10] Z. Benomar, F. Longo, G. Merlino, and A. Puliafito. Cloud-based network virtualization in iot with openstack. *ACM Trans. Internet Technol.*, 22(1), sep 2021.

[11] F. Callegati, W. Cerroni, and C. Contoli. Virtual networking performance in openstack platform for network function virtualization. *JECE*, 2016, apr 2016.

[12] J. Castillo-Lema, A. Venâncio Neto, F. de Oliveira, and S. Takeo Kofuji. Mininet-nfv: Evolving mininet with oasis tosca nvf profiles towards reproducible nfv prototyping. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 506–512, 2019.

[13] C. Cui, H. Deng, D. Telekom, U. Michel, and H. Damker. Network functions virtualisation.

[14] P. P. Deshmukh and S. Y. Amdani. Virtual memory management of private cloud using openstack api. In *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*, pages 1–4, 2019.

[15] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015.

[16] K. Katsalis, N. Nikaein, and A. Edmonds. Multi-domain orchestration for nfv: Challenges and research directions. In *2016 15th International Conference on Ubiquitous Computing and Communications and 2016 International Symposium on Cyberspace and Security (IUCC-CSS)*, pages 189–195, 2016.

[17] M. Mechtri, C. Ghribi, O. Soualah, and D. Zeghlache. Nfv orchestration framework addressing sfc challenges. *IEEE Communications Magazine*, 55(6):16–23, 2017.

[18] D. Muelas, J. Ramos, and J. E. L. d. Vergara. Assessing the limits of mininet-based environments for network experimentation. *IEEE Network*, 32(6):168–176, 2018.