

Design and Implementation of a Load Balancer in OpenStack

submitted by

Alapan Ghosh

Class Roll No: 002210503047

Exam Roll: MCA244046

Master of Computer Application

under the supervision of

Prof. Chandan Mazumdar

in the

Department of Computer Science and Engineering

Faculty of Engineering and Technology

Jadavpur University

Faculty of Engineering and Technology Jadavpur University

CERTIFICATE OF RECOMMENDATION

To Whom It May Concern

I hereby recommend that the project report entitled “Design and Implementation of a Load Balancer in OpenStack” has been carried out by Alapan Ghosh (Reg. No.: 163639 of 2022-23, Roll No.: 002210503047, Exam Roll: MCA244046), under my guidance and supervision and be accepted in partial fulfilment of the requirement for the degree of Master of COMPUTER APPLICATION in DEPARTMENT of COMPUTER SCIENCE and ENGINEERING, JADAVPUR UNIVERSITY during the academic year 2023-2024.

Prof. Chandan Mazumdar
Project Supervisor
Dept. of Computer Science and
Engineering
Jadavpur University

Prof. (Dr.) Nandini Mukhopadhyay
Head of the Department
Dept. of Computer Science & Engineering
Jadavpur University

Prof. Dipak Laha
Dean, Faculty Council
of Engineering & Technology
Jadavpur University, Kolkata-032

Faculty of Engineering and Technology

Jadavpur University

CERTIFICATE OF APPROVAL

This is to certify that the project report entitled “**Design and Implementation of a Load Balancer in OpenStack**” is a bonafide record of work carried out by ALAPAN GHOSH, in fulfillment of the requirements for the award of the degree of Master of Computer Application in the Department of Computer Science and Engineering, Jadavpur University. 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 project report only for the purpose for which it has been submitted.

Signature of Examiner 1

Date :

Signature of Examiner 2

Date :

Faculty of Engineering and Technology
Jadavpur University

**DECLARATION OF COMPLIANCE WITH
ACADEMIC ETHICS**

I hereby declare that this project report entitled "Design and Implementation of a Load Balancer in OpenStack" is an outcome of my genuine and independent work, as part of my Master of Computer Application program.

All information in this document has been obtained and presented in accordance with academic rules and ethical conduct.

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

Alapan Ghosh

Registration No.: 163639 of 2022-23

Examination Roll No.: MCA244046

Thesis Title :- Design and Implementation of a Load Balancer in OpenStack

Signature with Date

ACKNOWLEDGEMENT

With my most sincere and gratitude, I would like to thank Prof. Chandan Mazumdar, Department of Computer Science & Engineering, my supervisor, for his overwhelming support throughout the duration of the project. His motivation always gave me the required inputs and momentum to continue with my work, without which the project work would not have taken its current shape. His valuable suggestions and numerous discussions have always inspired new ways of thinking. I feel deeply honored that I got this opportunity to work under him.

I am particularly grateful to Dr. Srijita Basu, Dr. Anirban Sengupta, and the research scholars of CDC JU lab for their seamless cooperation and support throughout this project. Their assistance was instrumental in achieving its successful completion.

I would like to express my sincere thanks to all my teachers for providing sound knowledge base and cooperation.

I would like to thank all the faculty members of the Department of Computer Science & Engineering of Jadavpur University for their continuous support.

Last, but not the least, I would like to thank my batch mates for staying by my side when I needed them.

Alapan Ghosh

Class Roll No.: 002210503047

Examination Roll No.: MCA244046

Department of Computer Science & Engineering

Jadavpur University

ABSTRACT

In modern distributed computing environments, efficient load balancing is paramount for optimizing resource utilization and ensuring seamless service delivery. This research presents an innovative approach to load balancing that dynamically distributes incoming requests among servers based on their performance metrics. Weighted Round Robin assigns weights to servers according to their capacities and performance characteristics, enabling more proportional workload distribution and enhanced system performance.

OpenStack, as a versatile open-source cloud computing platform, forms the backbone of this architecture, providing the necessary infrastructure for implementing and managing cloud services. Additionally, the integration of a Virtual Network Function (VNF) within the OpenStack environment acts as a router, facilitating seamless communication between components and demonstrating the practical application of network function virtualization in enhancing network efficiency.

To further enhance system reliability, proactive measures are implemented to prevent server overload situations. Servers automatically stop accepting new requests when their resource utilization exceeds predefined thresholds, mitigating the risk of performance degradation or system failure. Additionally, a fail-safe mechanism is employed where the load balancer queues incoming requests during server unavailability, ensuring uninterrupted service delivery until servers are ready to accept requests again. These proactive measures bolster system resilience and maintain service continuity in the face of transient server failures or high demand periods.

Critical to the load balancing architecture is the integration of a Health Checker component, to provide real-time monitoring and visualization of server metrics within the terminal interface. Through these integrated components and advanced techniques, the proposed load balancing framework offers a robust solution for optimizing resource allocation, enhancing system reliability, and ensuring uninterrupted service availability in dynamic computing environments.

LIST OF FIGURES

Figure 01: LogicalArchitecture for an OpenStack cloud.....	13
Figure 02: Relationship among OpenStack services.....	15
Figure 03: NFV Framework and its components.....	21
Figure 04: Tacker Architecture.....	22
Figure 05: OpenStack Login Page.....	28
Figure 06: OpenStack Dashboard.....	29
Figure 07: Creating Image in OpenStack.....	31
Figure 08: List of all available images in OpenStack.....	32
Figure 09: Subnet Tab to Create Networks in OpenStack.....	33
Figure 10: Details of created VIM.....	35
Figure 11: VIM Management in OpenStack Horizon.....	36
Figure 12: New created VNFD in VNFD Catalog.....	38
Figure 13: Successful creation of VNF.....	39
Figure 14: Result of ifconfig in VNF's console.....	40
Figure 15: Instances Page in OpenStack Horizon.....	41
Figure 16: Creating Instances in OpenStack Horizon.....	42
Figure 17: Successful creation of all instances.....	43
Figure 18: Network Topology of the instances.....	44
Figure 19: Network Graph of the instances.....	44
Figure 20: Logical Architecture of Load Balancer along with Health Checker.....	46
Figure 21: Console of an Instance.....	53
Figure 22: Logging into an Instance.....	53
Figure 23: Successful initialisation of server.....	54
Figure 24: Providing details of servers to health checker.....	54

Figure 25: Health Checker Dashboard.....	55
Figure 26: Providing details of servers to load balancer	56
Figure 27: Load Balancer Dashboard.....	56
Figure 28: Logger in Load Balancer displaying forwarded requests.....	58
Figure 29: Replies received by Client from Load Balancer and Server.....	59
Figure 30: Health Checker Dashboard displaying that a Server is unavailable.....	59
Figure 31: Load Balancer Dashboard updating its status for unavailable server.....	60
Figure 32: Load Balancer Dashboard keeping track of size of Waiting Queue.....	60
Figure 33: Load Balancer Logger displaying message while clearing the Queue.....	60

LIST OF TABLES

Table 01: Sample Data of Servers.....49

Table 02: Normalized Data of Servers.....50

LIST OF ABBREVIATIONS

NFV: Network Function Virtualization

VNF: Virtual Network Function

API: Application Programming Interface

VM: Virtual Machine

NaaS: Network as a Service

SDN: Software-Defined Networking

LVM: Logical Volume Manager

REST: Representational State Transfer

SSL: Secure Sockets Layer

SLA: Service Level Agreement

IP: Internet Protocol

DDoS: Distributed Denial-of-Service

PNF: Physical Network Function

VNFM: Virtual Network Function Manager

NFVO: Network Function Virtualization Orchestrator

ETSI MANO: European Telecommunications Standards Institute - Management and Orchestration

VIM: Virtual Infrastructure Manager

VNFD: Virtual Network Function Descriptor

WRR: Weighted Round Robin (load balancing algorithm)

CONTENTS

ABSTRACT.....	6
LIST OF FIGURES.....	7
LIST OF TABLES.....	9
LIST OF ABBREVIATIONS.....	10
1. INTRODUCTION.....	12
1.1 Introduction to OpenStack.....	12
1.2 OpenStack Architecture and Components.....	13
1.3 Motivation.....	15
1.4 Structure of Thesis.....	16
2. INTRODUCTION TO LOAD BALANCING.....	17
2.1 Definition and Purpose.....	17
2.2 Load Balancing Measures.....	17
2.3 Load Balancing Algorithms.....	18
2.4 Challenges in Load Balancing.....	19
3. NETWORK FUNCTION VIRTUALIZATION.....	20
3.1 Virtual Network Functions (VNFs).....	20
3.2 Network Function Virtualization Architecture.....	21
3.3 Tacker: NFV Orchestration Service.....	22
4. SETUP AND CONFIGURATION.....	24
4.1 OpenStack installation with Devstack.....	24
4.2 Configuration of Cloud Images and Networks.....	30
4.3 Implementation of VNF.....	34
4.4 Configuration of Cloud Instances.....	40
5. IMPLEMENTING LOAD BALANCER.....	45
5.1 Custom Load Balancing Strategy.....	45
5.2 Calculation using Load Balancing Metrics.....	46
5.3 Dynamic Server Management and Status Updates.....	51
6. RESULT AND ANALYSIS.....	53
6.1 Initialising the Instances.....	53
6.2 Starting the Servers.....	54
6.3 Initialising the Health Checker.....	54
6.4 Initialising the Load Balancer.....	55
6.5 Sending Requests to Load Balancer.....	57
6.6 Analysis.....	58
7. CONCLUSION.....	61
REFERENCES.....	62

1. INTRODUCTION

1.1 Introduction to OpenStack

In July 2010, NASA and Rackspace Hosting jointly launched an open-source cloud software initiative known as OpenStack [1]. This initiative aimed to create a robust, scalable platform for cloud computing that could be freely available to the public and developers worldwide. OpenStack is governed under the Apache License, making it a free and open-source platform. It provides a comprehensive suite of tools for creating and managing private, public, and hybrid cloud environments [2].

OpenStack's software is designed to control a broad spectrum of processing, storage, and networking resources across a data center. It can be considered an Infrastructure as a Service (IaaS) model, with functionalities closely integrated with Platform as a Service (PaaS) characteristics [3]. This integration facilitates the deployment and management of various cloud services, enabling organizations to build and manage scalable cloud environments efficiently.

The platform comprises several interconnected components, each managing different aspects of cloud infrastructure. These components, referred to as "projects" within OpenStack, collectively provide comprehensive cloud services [4]. The core services include compute, networking, and storage, which are essential for any cloud infrastructure. Each project addresses specific needs and functionalities, ensuring that the platform can cater to diverse use cases and deployment scenarios.

OpenStack is an open-source, free cloud computing platform and it gives some benefits that's why OpenStack is a good Cloud computing platform. Some benefits are :

- **Easy Scalability:** Unlike physical servers with limited resources, OpenStack's virtual servers can readily scale up or down based on demand, ensuring we always have the resources we need.
- **Effortless Automation:** OpenStack simplifies cloud management through built-in automation tools. This reduces manual tasks and streamlines operations.
- **Rapid Development:** OpenStack, similar to Linux, fosters innovation. Its open-source nature allows for various distributions with unique features while maintaining a common core. This fosters competition and drives rapid development.
- **Strong Community:** OpenStack boasts a vibrant community of users and developers exceeding 4,000 contributors. This translates to ongoing support, knowledge sharing, and continuous improvement.
- **User-Friendly Management:** OpenStack's control panel provides clear visibility, control, and easy access to powerful management tools. This simplifies administration, resource monitoring, and user management.

1.2 OpenStack Architecture and Components

The OpenStack Logical Architecture illustrates the most common integrated services within OpenStack and their interactions. Fig. 1. shows the most common services of an OpenStack cloud. Users can interact with these services through the dashboard, command-line interfaces (CLIs), and APIs. All services communicate using open APIs, except in cases requiring privileged administrator commands.

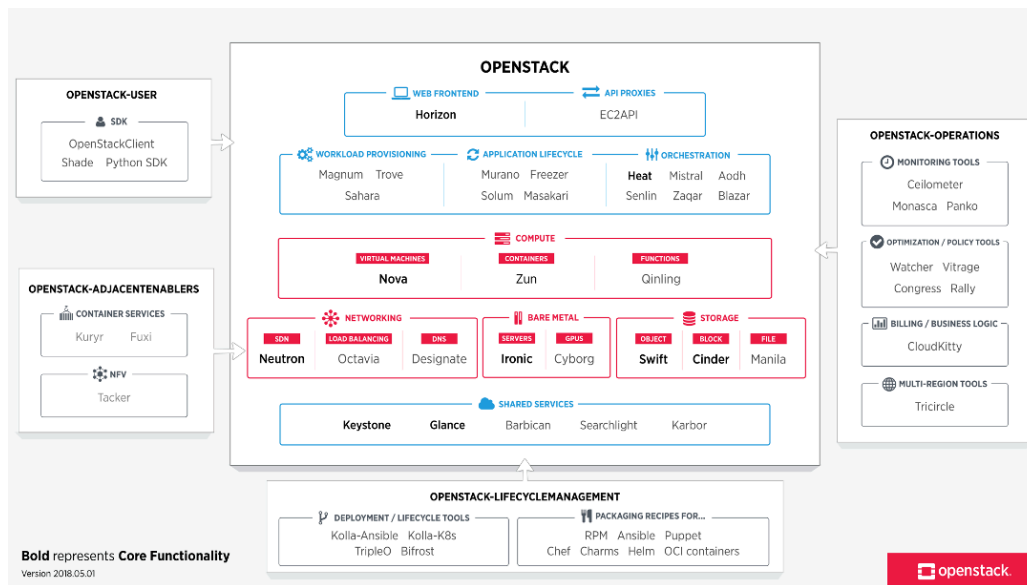


Fig. 1. Map of OpenStack projects[21]

Unlike traditional virtualization, where resources such as RAM and CPU are abstracted from the hardware using hypervisors, OpenStack employs a wide array of Application Programming Interfaces (APIs) to achieve this abstraction [5]. This API-driven approach allows users and administrators to interact directly with the cloud services, facilitating greater flexibility and control.

1.2.1 Core Components

1. **Compute (Nova):** Manages virtual machines (VMs) across various hypervisors (KVM, Xen, VMware), providing provisioning, scheduling, and resource optimization [9].
2. **Networking (Neutron):** Delivers Network as a Service (NaaS), enabling network creation, subnet management, and advanced features like firewalls and load balancing. It supports diverse networking technologies and integrates with SDN solutions [10].
3. **Storage:**
 - **Block Storage (Cinder):** Manages persistent block storage for VMs, allowing data persistence for applications. It supports various storage backends like LVM, Ceph, and commercial systems [11].

- **Object Storage (Swift):** Provides scalable and redundant storage for large datasets like backups and archives. Its distributed architecture ensures high availability and scales horizontally [14].
- 4. **Identity (Keystone):** Acts as the authentication and authorization service, managing users, roles, and permissions with multi-tenancy and RBAC support. It integrates with various authentication systems and provides a service catalog for service discovery [12].
- 5. **Image (Glance):** Handles VM image discovery, registration, and retrieval, facilitating VM deployment. It supports various image formats and integrates with storage backends for efficient image management [13].
- 6. **Dashboard (Horizon):** Provides a user-friendly web interface for managing cloud resources (instances, networks, storage) through an intuitive GUI. It offers features for launching instances, creating volumes, and configuring networks [16].
- 7. **Orchestration (Heat):** Enables automated deployment and management of cloud applications using templates. It defines infrastructure as code, automating resource creation and management for consistent deployments [15].
- 8. **Telemetry (Ceilometer):** Collects and stores usage and performance metrics from OpenStack services. This data is crucial for billing, benchmarking, and resource optimization decisions. It integrates with monitoring and analytics tools for enhanced cloud management [17].

1.2.2 Supporting Services

1. **Database Service (Trove):** Trove simplifies the management of databases in OpenStack, providing scalable and reliable database as a service (DBaaS). It supports multiple database engines, automating deployment and administrative tasks [18].
2. **Messaging Service (Zaqar):** Zaqar offers a multi-tenant cloud messaging service for OpenStack, enabling communication between distributed application components. It supports both message queues and notification services [19].
3. **Bare Metal Provisioning (Ironic):** Ironic facilitates the management and provisioning of physical machines, enabling OpenStack to handle both virtual and bare-metal servers. It is particularly useful for workloads requiring dedicated hardware resources [20].

OpenStack's architecture is designed around a modular and distributed approach, where each core component or service, often referred to as a "project," operates independently but integrates seamlessly with others through well-defined APIs. These services communicate with each other primarily through RESTful APIs, enabling interoperability and flexibility. Fig. 2. shows the relationships among the OpenStack services. For instance, when a user requests the creation of a new virtual machine through the Horizon dashboard, Horizon interacts with the Keystone service to authenticate the user and verify their permissions. Once authenticated, Horizon sends the request to the Nova service to provision the VM. Nova may

then communicate with Neutron to allocate networking resources and with Cinder to attach block storage to the VM if required. Each of these services, in turn, may interact with other components such as Glance to retrieve the VM image or Ceilometer to log the operation for monitoring and billing purposes [1][5].

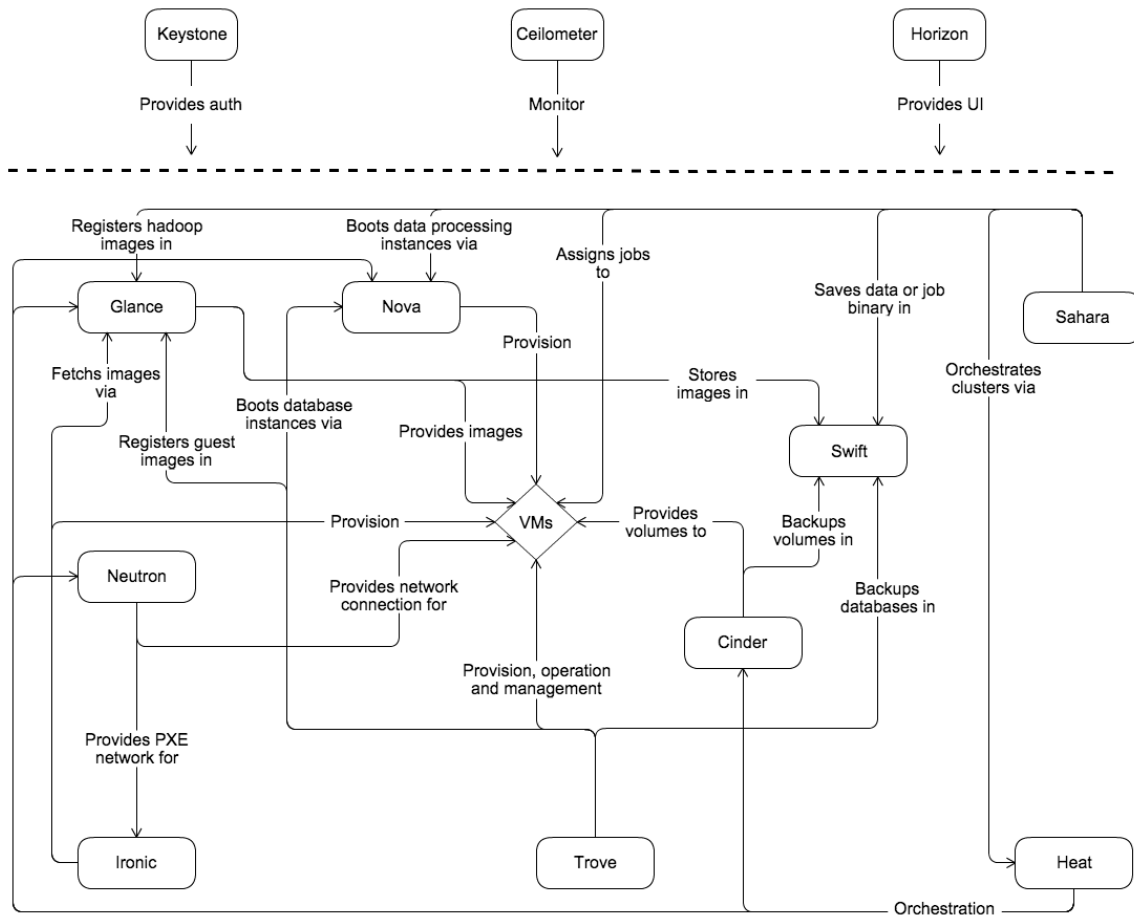


Fig. 2. Relationship among OpenStack services

This loosely coupled architecture allows each service to be scaled, upgraded, or replaced independently, ensuring high availability and scalability of the cloud infrastructure. The integration of these services forms a cohesive and powerful cloud ecosystem capable of managing a wide array of cloud resources efficiently [1].

1.3 Motivation

The primary motivation behind this research is to design and implement a cloud architecture using OpenStack that incorporates advanced load balancing techniques to optimize resource utilization and ensure high availability. By establishing an architecture with three servers, a load balancer, and a health checker, this project aims to demonstrate how effective load distribution and continuous monitoring can enhance the performance and reliability of cloud services.

In addition, the integration of a Virtual Network Function (VNF) within the OpenStack environment is a key component of this study. The VNF, acting as a router, facilitates seamless communication between the various components, thus demonstrating the practical application of NFV in modern cloud infrastructures. This integration not only highlights the versatility of OpenStack in supporting diverse network functions but also underscores the importance of NFV in improving network efficiency and reducing operational complexities.

The ultimate goal of this research is to contribute to the field of cloud computing by providing a practical and scalable solution for efficient resource management and network function virtualization using OpenStack.

1.4 Structure of Thesis

This project is organized into six sections, each focusing on different aspects of the research and implementation process. The structure guides the reader through the theoretical foundations, practical implementation, and analysis of the proposed cloud architecture using OpenStack.

The upcoming section defines load balancing and discusses its importance in optimizing resource utilization and ensuring high availability in distributed systems. It explores various metrics and algorithms used for load balancing and addresses common challenges in implementing effective load balancing solutions.

Following this, the section introduces the concept of Virtual Network Functions (VNFs) and their role in network function virtualization. It provides a detailed discussion on NFV architecture and its integration with OpenStack, along with an examination of Tacker, an NFV orchestration service.

The subsequent section offers a step-by-step guide on installing OpenStack using Devstack. It includes instructions on configuring cloud images and networks to support the deployment of VNFs, providing a practical foundation for setting up a functional NFV-enabled OpenStack environment.

The following section focuses on the implementation of a custom load balancing strategy. It covers the calculation of load balancing metrics, dynamic server management, and status updates to ensure efficient load distribution across the cloud infrastructure.

The final section presents the results and analysis of the implemented load balancing solution. It details the initialization of instances and servers, the activation of health checkers, and the load balancer's initialization.

This structured approach ensures a thorough exploration of integrating load balancing within an OpenStack environment, particularly in the context of NFV, offering practical and scalable solutions for efficient resource management and network function virtualization.

2. INTRODUCTION TO LOAD BALANCING

2.1 Definition and Purpose

Load balancing is a critical component in modern IT infrastructure, ensuring that workloads are distributed evenly across multiple servers or resources to optimize performance, reliability, and resource utilization. By balancing the load, systems can handle higher levels of traffic and improve overall efficiency, reducing the risk of any single server becoming a bottleneck.

Load balancing involves distributing incoming network traffic across multiple servers. This process helps to achieve redundancy, enhance performance, and ensure no single server is overwhelmed with too many requests. The primary goals of load balancing are to improve the availability and reliability of applications by distributing the workload across multiple resources, and to enhance the user experience by minimizing latency and maximizing throughput [6].

2.2 Load Balancing Measures

Effective load balancing requires the careful monitoring and measurement of various performance metrics to ensure that the system operates efficiently and reliably. The following measures are crucial in evaluating the performance of load balancers:

- **Server Utilization:** Server utilization measures how effectively the load balancer distributes the workload across all available servers. It is typically expressed as a percentage of the server's total capacity. Monitoring server utilization helps ensure that no single server is overburdened while others remain underutilized. Properly balanced server utilization leads to more efficient resource use, better performance, and extended hardware life [8].
- **Fault Tolerance:** Fault tolerance is the load balancer's ability to maintain service availability despite server failures or network issues. This measure ensures that the system can continue to operate smoothly even if one or more servers go offline. Load balancers achieve fault tolerance through techniques such as health checks, which regularly monitor server status and automatically reroute traffic to healthy servers. High fault tolerance is crucial for mission-critical applications where downtime can lead to significant financial losses or reputational damage [8].
- **Response Time:** Response time is the total time taken from when a client makes a request to when they receive a response. It includes both network latency and server processing time. Monitoring response time helps identify performance issues and ensures that the system meets the required service level agreements (SLAs). Load balancers improve response times by distributing requests to servers with the fastest response capabilities and by offloading tasks like SSL termination and caching [8].
- **Health Checks:** Health checks are periodic tests conducted by the load balancer to verify the status and performance of backend servers. These checks can include

simple ping tests, HTTP response checks, or more complex application-level tests. Effective health checks ensure that traffic is only directed to healthy and responsive servers, improving overall system reliability and user experience. Regular health checks are a critical measure for maintaining fault tolerance and high availability [8].

2.3 Load Balancing Algorithms

Load balancing algorithms are crucial in determining how incoming network traffic is distributed across multiple servers to ensure optimal performance, reliability, and resource utilization. Various algorithms are used depending on the specific requirements and goals of the application or service. Below are detailed descriptions of some common load balancing algorithms:

- **Round Robin:** The Round Robin algorithm distributes incoming requests sequentially across a pool of servers. Each server receives a request in turn, ensuring a simple and even distribution of the load. This method is straightforward to implement and works well in environments where servers have similar specifications and workloads. However, it does not consider the current load or capacity of the servers, which can lead to inefficiencies if servers vary significantly in performance or if they experience different load levels [37].
- **Least Connections:** The Least Connections algorithm directs incoming traffic to the server with the fewest active connections. This dynamic method helps balance the load more effectively, especially in environments where the duration of requests can vary significantly. By focusing on the current load rather than just distributing requests evenly, it helps prevent servers from becoming overwhelmed. This algorithm is particularly useful for applications where session durations are unpredictable and can vary widely [6].
- **IP Hash:** The IP Hash algorithm uses the client's IP address to determine which server should handle the request. By applying a hash function to the IP address, the algorithm ensures that a client consistently connects to the same server, which can be beneficial for session persistence. This method helps maintain a consistent user experience, as data related to the user's session remains on the same server. However, it can lead to uneven load distribution if the client base is not evenly spread across the hash space [6].
- **Weighted Round Robin:** Weighted Round Robin is an enhanced version of the Round Robin algorithm. In this method, each server is assigned a weight based on its capacity or performance capabilities. Servers with higher weights receive a proportionally larger share of the incoming requests. This algorithm is useful in heterogeneous environments where servers have different processing powers or capabilities. By considering the weights, it ensures that more powerful servers handle a greater portion of the traffic, optimizing resource utilization [37].

- **Weighted Least Connections:** Similar to Weighted Round Robin, the Weighted Least Connections algorithm assigns weights to servers based on their capacity. However, instead of distributing traffic sequentially, it directs traffic to servers with the least number of active connections while considering their weights. This method combines the benefits of least connections and weighted distribution, providing a more balanced and efficient load distribution in environments with servers of varying capacities and unpredictable request durations [6].
- **Least Response Time:** The Least Response Time algorithm directs traffic to the server with the lowest average response time. This method aims to reduce latency and improve user experience by considering the speed at which servers can handle requests. It is particularly useful for applications where response time is critical, such as real-time services. By constantly monitoring and comparing response times, the load balancer can dynamically adjust the traffic distribution to optimize performance [6].
- **Random:** The Random algorithm distributes incoming requests to servers randomly. While simple, this method does not consider the current load or capacity of the servers. It can be useful in small, homogeneous environments where all servers have similar capabilities. However, it may lead to inefficient load distribution in larger or more complex environments, as it does not account for server performance or current load [6].

2.4 Challenges in Load Balancing

Some of the challenges faced during load balancing are as follows:

1. **Scalability:** Ensuring the load balancer can handle increasing amounts of traffic without compromising performance. As applications grow, the load balancing infrastructure must scale accordingly [8].
2. **Configuration Complexity:** Setting up and managing load balancers, especially in complex environments, can be challenging. Proper configuration is essential for optimal performance and security [8].
3. **Security:** Load balancers must protect against various cyber threats while managing traffic efficiently. This includes defending against DDoS attacks and ensuring secure data transmission [8].
4. **Consistency:** Maintaining session persistence, also known as sticky sessions, where a user's requests are consistently routed to the same server, can be difficult but is necessary for certain applications [8].

3. NETWORK FUNCTION VIRTUALIZATION

3.1 Virtual Network Functions (VNFs)

A Network Function (NF) is a network component characterized by well-defined external interfaces and functional behavior. Examples of NFs include DHCP, Firewall, SAE Gateway, MME, and eNodeB. A Virtual Network Function (VNF) is the software version of a network function that can be quickly and easily deployed on a virtual resource, such as a virtual machine (VM) . A service provided by a service provider typically includes one or more VNFs.

The virtualization of an NF does not affect its functional behavior or status. Both Physical Network Functions (PNFs) and VNFs are expected to have identical functional behavior and external operational interfaces . A VNF may comprise various internal components. For example, one VNF might be distributed across multiple VMs, each hosting a different VNF component. Alternatively, an entire VNF can be deployed within a single VM .

A single VNF may include several internal components, such as firewalls, residential gateways, and packet data network gateways (PGWs), to simplify deployment and management. However, a VNF may also contain only one component to maximize scalability and reuse, and to ensure a quicker response time due to its simplicity. It is important to note that a single VNF can be deployed and distributed across numerous VMs [24].

From the perspective of Virtual Network Functions (VNFs), the hardware resources and virtualization layer appear as a unified entity that provides the necessary virtualized resources. NFVI (Network Function Virtualization Infrastructure) comprises three primary components:

1. **Hardware Resources:** These include standard computing, storage, and networking equipment that provide the foundation for VNFs [25].
2. **Virtualization Layer:** This software layer abstracts the physical hardware, allowing VNFs to run independently of the underlying infrastructure. It manages resource allocation and provisioning for VNFs [25].
3. **Virtualized Resources:** The virtualization layer transforms physical resources into virtual ones (CPU, memory, storage) that VNFs can utilize [25].

Benefits of VNFs [24]:

- Faster service roll-out
- Improved scalability and elasticity
- Reduced hardware costs
- Greater flexibility in managing network services

3.2 Network Function Virtualization Architecture

NFV (Network Function Virtualization) integration within OpenStack involves harnessing its suite of components and services to deploy and manage virtualized network functions. This approach seeks to replace traditional hardware-based network appliances with software-based counterparts, executed as virtual machines within the OpenStack environment. The NFV Architecture, as outlined by ETSI, comprises three primary components:

- **NFVI (Network Function Virtualization Infrastructure):** The foundation for NFV, it includes [25]:
 - Physical and virtual resources (compute, storage, networking)
 - Virtualization capabilities provided by OpenStack
- **Virtual Network Functions (VNFs):** Software versions of traditional network appliances like firewalls and routers. VNFs are deployed as VMs within OpenStack, leveraging its resources [24].
- **NFV Management and Orchestration (MANO):** Manages the lifecycle of VNFs [26]:
 - NFVO (Orchestrator): Coordinates VNF deployment and configuration based on service requests.
 - VIM (Infrastructure Manager): Manages virtual resources and interfaces with VNFs.
 - VNFM (Function Manager): Handles individual VNF lifecycle management.

These components work in concert to establish a flexible and scalable NFV framework within OpenStack, enabling the dynamic deployment and management of virtualized network functions to meet evolving network requirements. Fig. 3. depicts the overarching NFV framework, illustrating the interaction between NFVI, VNFs, and MANO components.

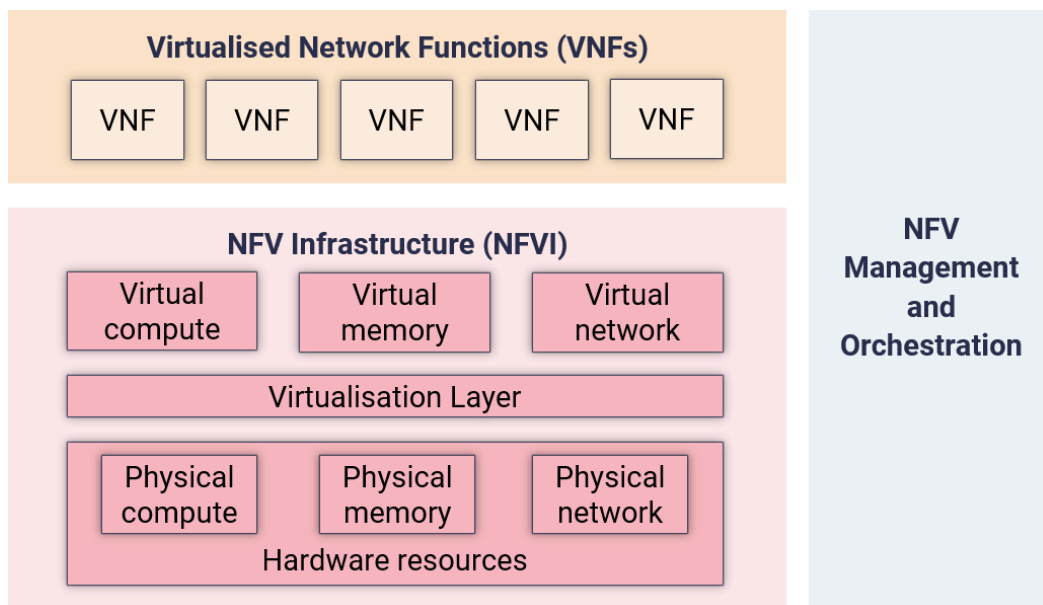


Fig. 3. NFV Framework and its components[23]

3.3 Tacker: NFV Orchestration Service

To deploy and manage Network Services and Virtual Network Functions (VNFs) on an NFV infrastructure platform like OpenStack, Tacker is an official OpenStack project dedicated to developing a Generic VNF Manager (VNFM) and an NFV Orchestrator (NFVO). Tacker provides a comprehensive functional stack designed to orchestrate network services end-to-end by utilizing VNFs. Fig. 4. shows the architecture of Tacker. It adheres to the ETSI MANO Architectural Framework, ensuring standardization and interoperability across different NFV implementations [22].

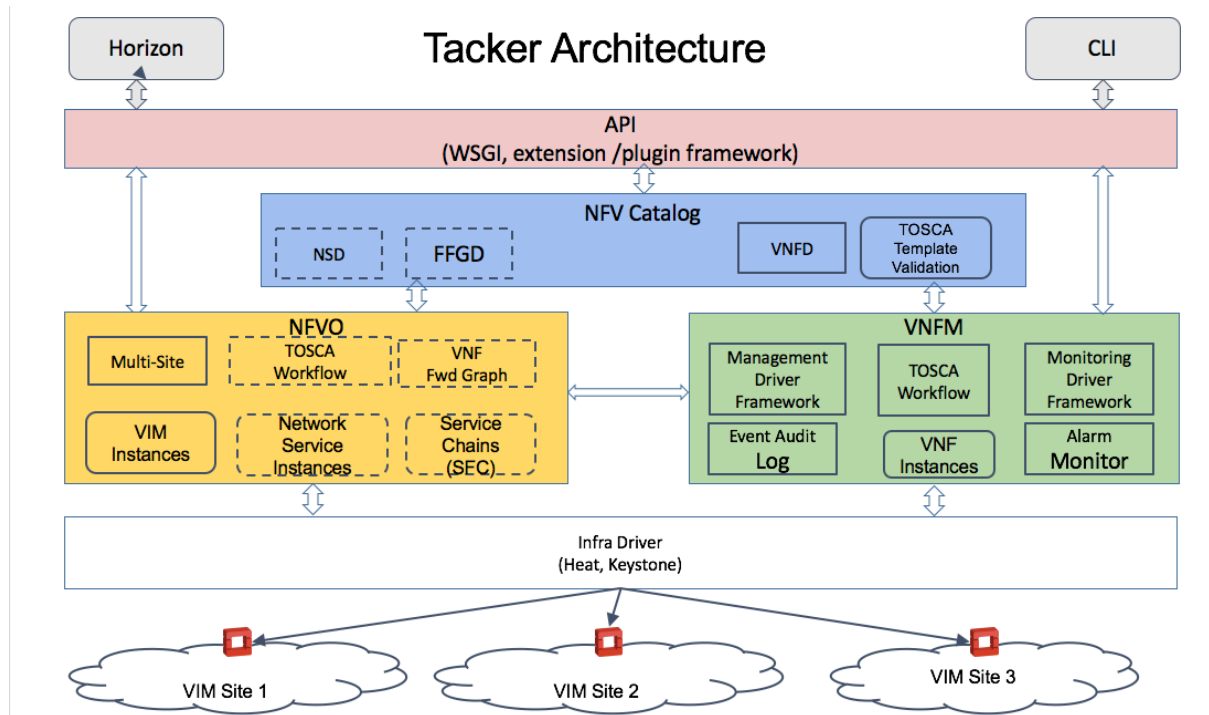


Fig. 4. Tacker Architecture[22]

Tacker comprises several key components, each playing a vital role in its overall functionality:

1. NFV Catalog:

- **VNF Descriptors:** Define VNF characteristics and requirements (configurations, resources) [22].
- **Network Services Descriptors:** Describe how VNFs interact to deliver a service [22].
- **VNF Forwarding Graph Descriptors:** Outline data flow paths between VNFs using Service Function Chaining (SFC) [22].

2. VNFM (Virtual Network Function Manager):

- **VNF Lifecycle Management:** Creates, updates, and deletes VNFs [22].
- **Placement:** Positions VNFs on optimal resources for performance [22].
- **Health Monitoring:** Tracks VNF health and identifies issues [22].

- **Auto Healing/Scaling:** Automatically recovers or adjusts VNF instances based on policies [22].
- **Initial Configuration:** Assists with initial VNF setup [22].
- 3. **NFVO (Network Function Virtualization Orchestrator):**
 - **Templated Deployment:** Uses templates to simplify and standardize service deployment [22].
 - **VNF Placement Policy:** Optimizes VNF placement based on defined policies [22].
 - **Service Function Chaining (SFC):** Connects VNFs according to network service requirements [22].
 - **VIM Resource Management:** Checks and allocates resources from the VIM before deploying VNFs [22].
 - **Multi-VIM/Site Orchestration:** Orchestrates VNFs across multiple VIMs and locations [22].
- 4. **VIM (Virtualized Infrastructure Manager):**
 - **Resource Management:** Allocates and manages virtualized compute, storage, and networking resources [22].
 - **Virtualization Abstraction:** Hides physical infrastructure complexity for easier VNF deployment [22].
 - **Integration with Orchestration:** Interfaces with NFVO and VNFM for automated VNF management [22].

In essence, Tacker provides a comprehensive framework for managing the lifecycle of VNFs within an NFV environment.

As OpenStack's Tacker project transitions away from legacy functionalities, exploring alternative approaches for Network Function Virtualization (NFV) management becomes crucial. This includes utilizing the evolving ETSI NFV-SOL functionalities within Tacker itself, leveraging OpenStack's existing orchestration tool, HEAT, for basic VNF deployment, or considering external Network Function Virtualization Orchestrators (NFVO) like Cloudify and OpenDaylight ODL for more comprehensive management capabilities. The optimal approach will depend on the specific requirements for VNF management and integration within the OpenStack environment.

4. SETUP AND CONFIGURATION

4.1 OpenStack installation with Devstack

In our lab, we've undertaken the implementation of Load Balancer using a consolidated approach, employing a single all-in-one node serving as the controller, compute, and storage unit.

Our initial step in deploying OpenStack involved creating a virtual machine (VM) within Proxmox, equipped with 16GB of RAM, 100GB of memory, and boasting 16 cores, alongside a Bridge network adapter. Proxmox Virtual Environment (PVE) serves as the backbone for managing virtualization tasks, functioning as a Type-2 hypervisor compatible with both Linux and Windows operating systems on x64 hardware. Rooted in Debian but tweaked with an Ubuntu LTS kernel, PVE streamlines the deployment and oversight of virtual machines and containers.

Subsequently, we proceeded to install Ubuntu 20.04 LTS DESKTOP OS on the designated VM, establishing a robust foundation for our OpenStack framework [27].

```
sudo apt-get update
```

Executing the above command retrieves the most recent version of the package list from our current distribution's software repository, as well as any third-party repositories that we have configured. Essentially, it determines the latest version of each package and its dependencies, without initiating the download or installation of any updates [27].

```
sudo apt-get install build-essential
```

The "build-essential" package is a crucial component within the Debian ecosystem. It doesn't function as standalone software; rather, it serves as a collection of essential packages necessary for creating Debian packages (deb). These essential packages include libc, gcc, g++, make, dpkg-dev, and others. When we install the "build-essential" package, it automatically installs all of these required packages in one streamlined command, ensuring that developers have everything they need to compile and build software within the Debian environment [27].

```
sudo apt-get install net-tools
```

This package encompasses vital tools for managing the network subsystem of the Linux kernel. Among these tools are arp, ifconfig, netstat, rarp, nameif, and route, which play crucial roles in configuring and troubleshooting network connections. Additionally, the package includes utilities tailored to specific network hardware types such as plipconfig, slattach, and mii-tool. It also offers advanced functionalities for IP configuration, including iptunnel and ipmaddr, allowing for comprehensive control and optimization of network settings and connectivity [27].

```
sudo apt-get install python3-pip
```

Pip serves as a package management system designed to streamline the installation and administration of software packages authored in Python. It facilitates the handling of packages available in the Python Package Index (PyPI), simplifying the process of discovering, acquiring, and managing Python libraries and tools. With Pip, users can effortlessly install, upgrade, and uninstall Python packages, enhancing the efficiency and ease of Python software development and deployment.

```
pip install -U os-testr A testr
```

This wrapper is designed to offer essential functionality for OpenStack projects. It serves as a layer that encapsulates and provides convenient access to the underlying features and services required by various components within the OpenStack ecosystem. By abstracting complex operations and interactions, the wrapper simplifies the development, deployment, and management of OpenStack projects, fostering greater efficiency and flexibility in building cloud computing solutions [27].

```
sudo apt-get install autoconf automake gdb git libffi-dev  
zlib1g-dev libssl-dev -y
```

This command installs several development and debugging tools: Autoconf helps generate configuration scripts for software projects, Automake creates Makefiles based on project specifications, and gdb is a powerful debugger for various programming languages.

Git, a widely used version control system, excels in managing large projects efficiently and swiftly. It underpins numerous prominent open-source endeavors, notably the Linux kernel. Git is renowned for its speed, scalability, and distributed nature. The libffi-dev package equips programs using libffi with necessary headers and static library files for building purposes. zlib, a library implementing the deflate compression method, powers gzip and PKZIP. This package includes development support files. libssl-dev, an integral part of the OpenSSL project, implements SSL and TLS cryptographic protocols for secure Internet communication.

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

This command creates a new user named stack to the system. The user will access the root opt directory [27].

```
sudo chmod +x /opt/stack
```

This command provides the execute permission to the directory [27].

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

No root password will be needed for the user stack.

```
sudo su - stack
```

This gives stack user as sudo user privilege also all the environment variables will be sourced to the user, stack [27].

```
git clone <https://opendev.org/openstack/devstack> -b
stable/yoga
```

This will clone the devstack folder from the link and it will clone the stable yoga branch [27].

```
cd devstack
```

Change the current directory to the devstack folder.

```
git branch
```

OpenStack version can be checked by this command. `nano local.conf`

Create a local.conf file with all the credentials and the services that are needed to be installed.

The provided local.conf file has been utilized in our laboratory for installing OpenStack through devstack:

```
[[local|localrc]]
HOST_IP=127.0.0.1
ADMIN_PASSWORD=devstack
MYSQL_PASSWORD=devstack
RABBIT_PASSWORD=devstack
SERVICE_PASSWORD=$ADMIN_PASSWORD
SERVICE_TOKEN=devstack
#####
# Customize the following section based on your installation
#####
# Pip
PIP_USE_MIRRORS=False
USE_GET_PIP=1
#OFFLINE=False
#RECLONE=True
# Logging
LOGFILE=$DEST/logs/stack.sh.log
VERBOSE=True
ENABLE_DEBUG_LOG_LEVEL=True
ENABLE_VERBOSE_LOG_LEVEL=True
```

```

# Disable security groups
LIBVIRT_FIREWALL_DRIVER=nova.virt.firewall.NoopFirewallDriver

# Enable heat, networking-sfc, barbican and mistral
enable_plugin heat https://opendev.org/openstack/heat
stable/yoga
enable_plugin networking-sfc
https://opendev.org/openstack/networking-sfc stable/yoga
enable_plugin barbican https://opendev.org/openstack/barbican
stable/yoga
enable_plugin mistral https://opendev.org/openstack/mistral
stable/yoga
# Ceilometer
#CEILOMETER_PIPELINE_INTERVAL=300
CEILOMETER_EVENT_ALARM=True
enable_plugin ceilometer
https://opendev.org/openstack/ceilometer stable/yoga
enable_plugin aodh https://opendev.org/openstack/aodh
stable/yoga
# Blazar
enable_plugin blazar https://github.com/openstack/blazar.git
stable/yoga
# Tacker
enable_plugin tacker https://opendev.org/openstack/tacker
stable/yoga
[[post-config|/etc/neutron/dhcp_agent.ini]]
[DEFAULT]
enable_isolated_metadata = True
Implementation and deployment of VNF
./stack

```

Executing `stack.sh` triggers the execution of a shell script that embodies an opinionated approach to installing OpenStack for developers. This script orchestrates the installation and configuration of diverse combinations of OpenStack services, including Ceilometer, Cinder, Glance, Heat, Horizon, Keystone, and Nova [27].

The script typically takes around 45 to 50 minutes to execute. However, it's susceptible to errors, particularly if there are network availability issues. In case of an error, it's necessary to execute `./unstack.sh`, which halts all processes initiated by `stack.sh`. Subsequently, running `./stack.sh` again restarts the installation process, attempting to overcome any previous issues and successfully complete the installation of OpenStack services. This cycle ensures that any encountered errors are addressed promptly, allowing for a smoother installation experience.

Upon successful installation, the OpenStack dashboard can be accessed by navigating to "http://<host_ip_address>/dashboard" in a web browser. Simply replace "<host_ip_address>" with the actual IP address of the host where OpenStack is installed. This URL will direct us to the dashboard interface, providing access to the various OpenStack services and functionalities through a user-friendly web interface. Fig. 5. shows how the login page of dashboard looks

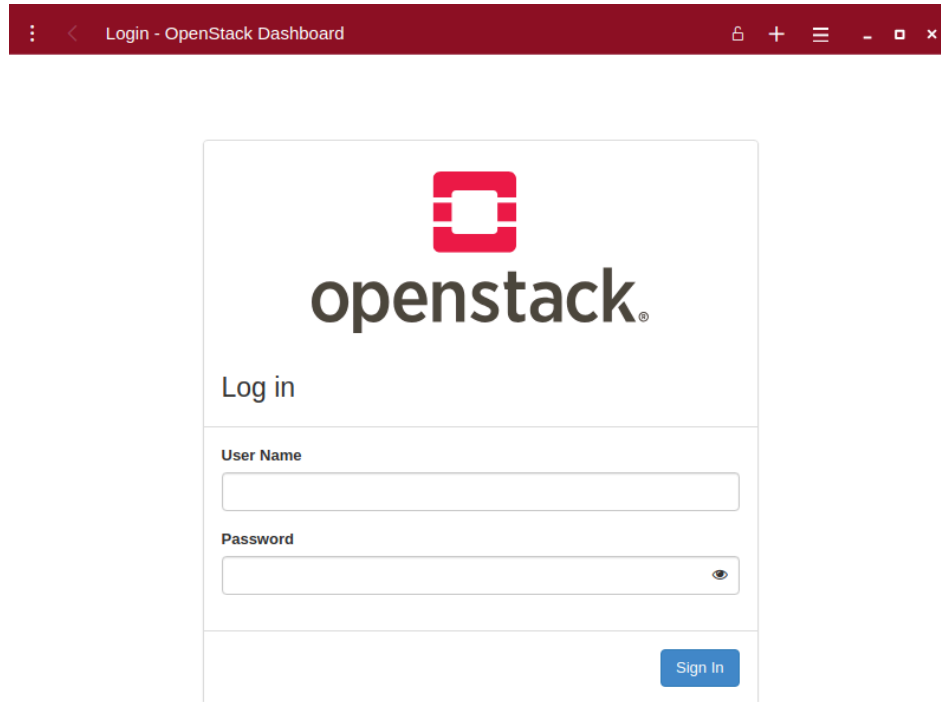


Fig. 5. OpenStack Login Page

Now we can login with the password given in local.conf file. Fig. 6. shows the dashboard after logging in.

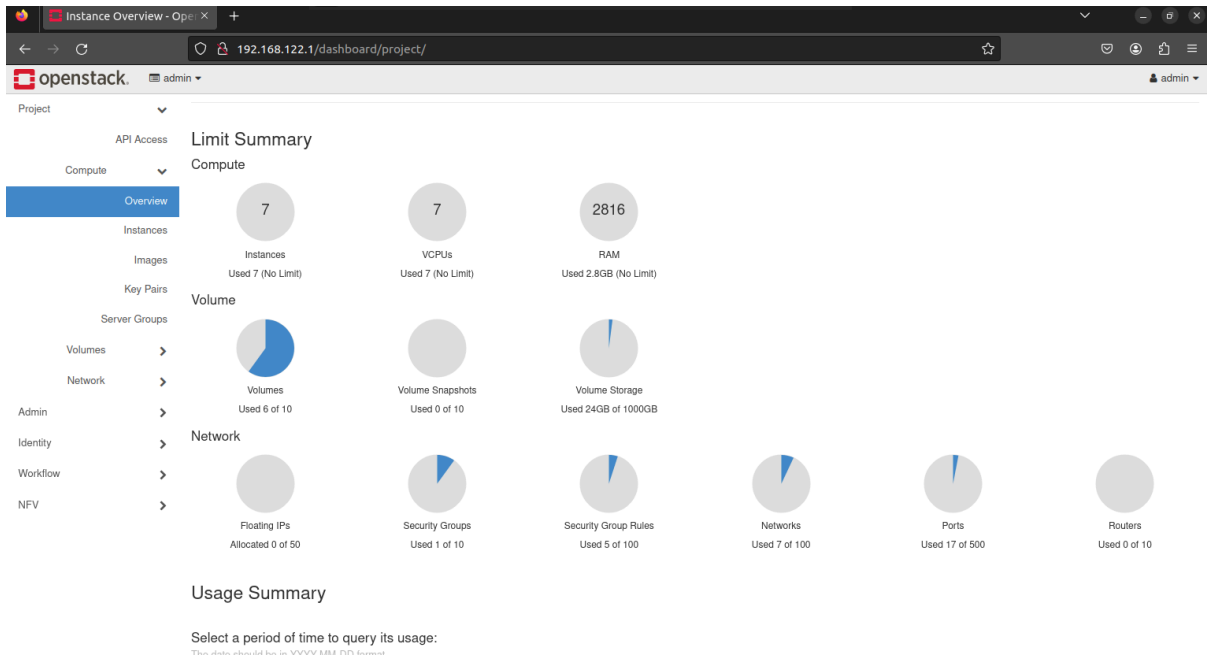


Fig. 6. Dashboard in OpenStack Horizon

To initiate OpenStack after a system restart, follow these steps:

Grant sudo privileges to the "stack" user and ensure all environment variables are sourced:

```
sudo su - stack
```

This command elevates the "stack" user to sudo privileges and loads all necessary environment variables.

Navigate to the devstack directory:

```
cd devstack
```

Changing the current directory to the devstack folder allows us to execute OpenStack-related commands and scripts from within that directory.

```
source openrc admin admin
```

This command loads the environment variables required for administrative tasks and sets the authentication context to the admin user and admin project, enabling the execution of administrative commands and operations within the OpenStack environment [27].

```
sudo systemctl start devstack@*
```

This command triggers the systemd service manager to start all services prefixed with "devstack@", implying all components installed and managed by devstack. This ensures that

the entire OpenStack infrastructure is up and running, enabling users to access and utilize its functionalities [27].

4.2 Configuration of Cloud Images and Networks

We install `virt-customize` and then use it to install Node.js inside a Linux image. Here are the detailed steps:

Install `virt-customize`

`virt-customize` is a tool from the `libguestfs` suite that allows us to customize virtual machine disk images. Install it using the following command [28]:

```
sudo apt-get update sudo apt-get install libguestfs-tools
```

Customize the Linux Image

Assume we have a Linux image file named `ubuntu.img`. We'll use `virt-customize` to install Node.js inside this image.

a. Download the Linux Image

If we don't already have a Linux image, we can download one. For example, to download an Ubuntu cloud image [28]:

```
wget<https://cloud-images.ubuntu.com/bionic/current/bionic-server-cloudimg-amd64.img> -O ubuntu.img
```

b. Use `virt-customize` to Install Node.js

Run the following command to install Node.js inside the image [28][29]:

```
sudo virt-customize -a ubuntu.img --update --install  
nodejs, npm
```

This command will:

- Update the package list inside the image.
- Install Node.js and npm (Node.js package manager) inside the image

c. Install the "blessed" Package:

```
sudo virt-customize -a ubuntu.img --run-command 'npm install  
-g blessed'
```

d. Verify the Installation:

```
sudo virt-customize -a ubuntu.img --run-command 'npm list -g --depth=0'
```

To create a Cloud Image in OpenStack using the customized `ubuntu.img` file, follow these steps:

First, log in to our OpenStack environment and navigate to the Horizon dashboard or use the OpenStack CLI to upload the image [30].

1. Log in to the OpenStack Horizon dashboard.
2. Navigate to the "Project" tab.
3. Under the "Compute" section, select "Images."
4. Click the "Create Image" button. We will see a box as in Fig. 7.

Fig. 7. Creating Image in OpenStack

5. Fill in the details for our new image:
 - **Name:** Provide a name for our image
 - **Image Source:** Select "Image File."
 - **Format:** Choose QCOW2 as the format of our image file.
 - **File:** Upload the `ubuntu.img` file.
6. Click "Create Image."

On successful creation we can see the image on the Images tab as in Fig. 8.

Images

Click here for filters or full text search. ✕ + Create Image Delete Images								
Displaying 5 items								
☐	Owner	Name ^	Type	Status	Visibility	Protected	Disk Format	Size
☐	admin	blessed	Image	Active	Shared	No	QCOW2	1.20 GB
☐	admin	cirros-0.5.2-x86_64-disk	Image	Active	Public	No	QCOW2	15.55 MB
☐	admin	cirros-0.6.2-x86_64-disk	Image	Active	Public	No	QCOW2	20.44 MB
☐	admin	Fedora-Cloud-Base-37-1.7.x86_64	Image	Active	Public	No	QCOW2	470.00 MB
☐	admin	OpenWRT	Image	Active	Public	No	RAW	3.73 MB
Displaying 5 items								

Fig. 8. List of all available Images in OpenStack

Initially we have network net0 with Network Address 10.10.0.0/24 and net1 with Network Address 10.10.1.0/24. To create the networks net2 and net3 with subnets 10.10.2.0/24 and 10.10.3.0/24 using the Horizon dashboard [31]:

- **Navigate to the Network Panel:**
 - In the left sidebar, go to the "Project" tab.
 - Under the "Network" section, click on "Networks."
- **Create Network net2:**
 - Click the "Create Network" button.
 - In the "Create Network" dialog, fill in the details for net2:
 - **Network Name:** net2
 - Click "Next" to go to the Subnet tab. It looks like as shown in Fig. 8.

Create Network

Network Subnet Subnet Details

Subnet Name
private-subnet

Network Address ⓘ
192.168.0.1/24

IP Version
IPv4

Gateway IP ⓘ

☐ Disable Gateway

Creates a subnet associated with the network. You need to enter a valid "Network Address" and "Gateway IP". If you did not enter the "Gateway IP", the first value of a network will be assigned by default. If you do not want gateway please check the "Disable Gateway" checkbox. Advanced configuration is available by clicking on the "Subnet Details" tab.

Cancel « Back Next »

Fig. 9. Subnet Tab to Create Networks in OpenStack

- Fill in the subnet details for **net2**:
 - **Subnet Name:** net2-subnet
 - **Network Address:** 10.10.2.0/24
- Click "Create" to create the network and its subnet.
- **Create Network net3:**
 - Click the "Create Network" button again.
 - In the "Create Network" dialog, fill in the details for **net3**:
 - **Network Name:** net3
 - Click "Next" to go to the Subnet tab.
 - Fill in the subnet details for net3:
 - **Subnet Name:** net3-subnet
 - **Network Address:** 10.10.3.0/24
 - Click "Create" to create the network and its subnet.

4.3 Implementation of VNF

After starting OpenStack, the next step is to create a Virtual Infrastructure Manager (VIM) to facilitate the implementation and deployment of Virtual Network Functions (VNFs) in OpenStack. Follow these steps:

Change directory to the tool folder in the tacker directory:

```
cd tacker/tools
```

Generate the `vim_config.yaml` file and initialize environment variables by executing `gen_vim_config.sh` [33]:

```
./gen_vim_config.sh
```

Copy the generated `vim_config.yaml` file to the devstack directory [33]:

```
cp vim_config.yaml ~/devstack
```

Navigate to the devstack directory:

```
cd ~/devstack
```

Register the default OpenStack VIM using the `vim_config.yaml` file [32]:

```
openstack vim register --config-file ./vim_config.yaml  
--is-default --description 'vim for nfv_user in nfv'  
openstack_nfv_vim
```

This command registers the default OpenStack VIM with the specified description. Once completed, we will have successfully configured the VIM for deploying and managing VNFs within the OpenStack environment. Fig. 9. shows the details of the created VIM.

Field	Value
auth_cred	{ "username": "admin", "user_domain_name": "default", "cert_verify": "True", "project_id": null, "project_name": "admin", "project_domain_name": "default", "auth_url": "http://127.0.0.1/identity/v3", "key_type": "barbican_key", "secret_uuid": "****", "password": "****" }
auth_url	http://127.0.0.1/identity/v3
created_at	2024-05-26 12:12:12.996733
description	vim for nfvi_user in nfvi
extra	
id	3ef9887f-a58c-4b3d-a045-5aea561e559c
is_default	True
name	openstack-nfv-vim
placement_attr	{ "regions": ["RegionOne"] }
project_id	0c52f2a0dbba46b0b66707543819f56b
status	ACTIVE
type	openstack
updated_at	None
vim_project	{ "name": "admin", "project_domain_name": "default" }

Fig. 10. Details of created VIM

Now that the VIM has been successfully registered, we can proceed to deploy Virtual Network Functions (VNFs). The initial step is to onboard the VNF descriptor (VNFD) file. In Fig. 10, we can see the dashboard displaying the deployed VIM.

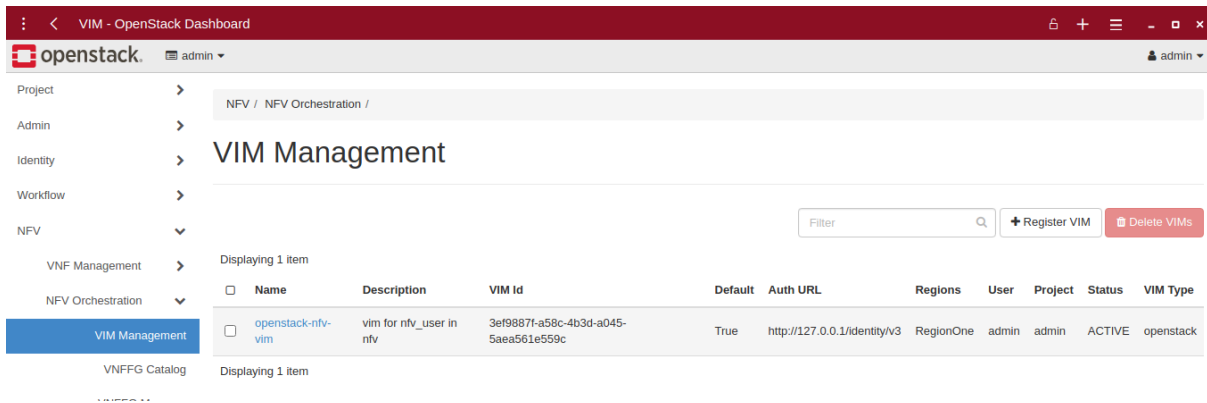


Fig. 11. VIM Management in OpenStack Horizon

To create a VNFD (Virtual Network Function Descriptor) file named "make_vnfd.yaml" using the TOSCA (Topology and Orchestration Specification for Cloud Applications) format, we can use a text editor like Nano. Below is the YAML code we can write in the "make_vnfd.yaml" file to define the VNFD:

```
tosca_definitions_version: tosca_simple_profile_for_nfv_1_0_0
description: Router for load balancer
metadata:
template_name: sample-vnfd-userdata
topology_template:
node_templates:
VDU1:
  type: tosca.nodes.nfv.VDU.Tacker
  capabilities:
  nfv_compute:
  properties:
  num_cpus: 1
  mem_size: 512 MB
  disk_size: 16 GB
  properties:
  image: blessed
  config: |
  param0: key1
  param1: key2
  mgmt_driver: noop
  user_data_format: RAW
  CP1:
    type: tosca.nodes.nfv.CP.Tacker
    properties:
    management: true
    ip_address: 10.10.0.1
    order: 0
```

```

    anti_spoofing_protection: false
    requirements:
    - virtualLink:
node: VL1
    - virtualBinding:
node: VDU1
VL1:
    type: toska.nodes.nfv.VL
    properties:
    network_name: net0
    vendor: Tacker
CP2:
    type: toska.nodes.nfv.CP.Tacker
    properties:
    management: true
    ip_address: 10.10.1.1
    order: 0
    anti_spoofing_protection: false
    requirements:
    - virtualLink:
node: VL2
    - virtualBinding:
node: VDU1
VL2:
    type: toska.nodes.nfv.VL
    properties:
    network_name: net1
    vendor: Tacker
CP3:
    type: toska.nodes.nfv.CP.Tacker
    properties:
    management: true
    ip_address: 10.10.2.1
    order: 0
    anti_spoofing_protection: false
    requirements:
    - virtualLink:
node: VL3
    - virtualBinding:
node: VDU1
VL3:
    type: toska.nodes.nfv.VL
    properties:
    network_name: net3

```

```

    vendor: Tacker
CP4:
    type: tosca.nodes.nfv.CP.Tacker
    properties:
        management: true
        ip_address: 10.10.3.1
        order: 0
        anti_spoofing_protection: false
        requirements:
            - virtualLink:
                node: VL4
            - virtualBinding:
                node: VDU1
VL4:
    type: tosca.nodes.nfv.VL
    properties:
        network_name: net4
        vendor: Tacker

```

To onboard the TOSCA VNFD template "tosca-vnfd-userdata.yaml" to the Tacker VNFD Catalog and create a VNFD named "ud" (Security Group), we can use the following command [22]:

```
tacker vnfd-create --vnfd-file toska-vnfd-userdata.yaml ud
```

This command instructs Tacker to create a VNFD with the name "ud" using the TOSCA VNFD template provided in the "tosca-vnfd-userdata.yaml" file.

Upon successful execution of the command, we should see a confirmation message indicating that the VNFD has been created successfully. In the provided Fig. 11. we can observe the successful creation of the VNFD in the Tacker VNFD Catalog.

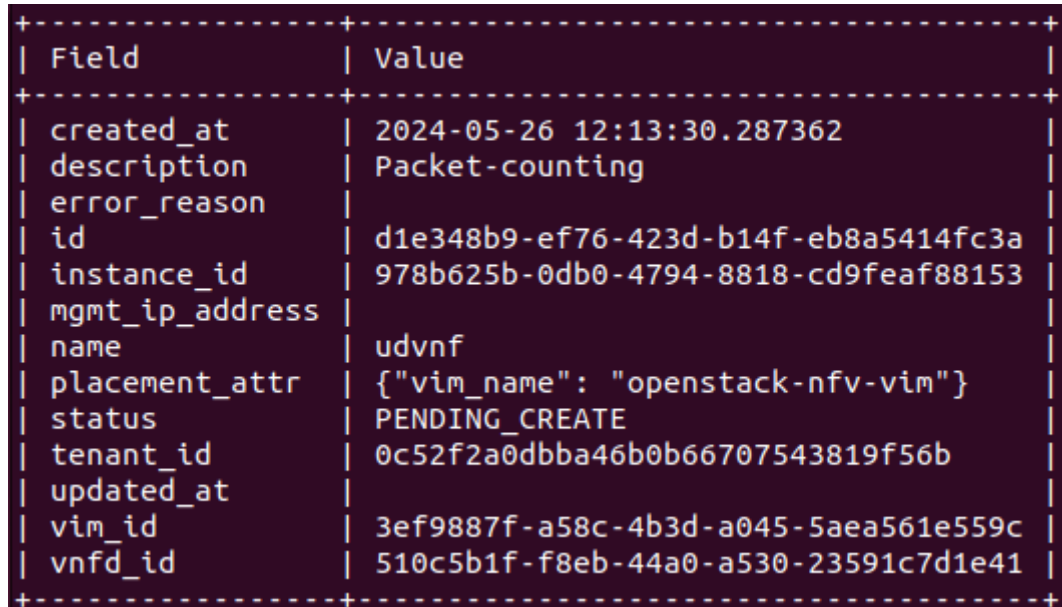
Field	Value
created_at	2024-05-26 12:13:10.137305
description	Packet-counting
id	510c5b1f-f8eb-44a0-a530-23591c7d1e41
name	ud
service_types	vnfd
template_source	onboarded
tenant_id	0c52f2a0dbba46b0b66707543819f56b
updated_at	

Fig 12. New created VNFD in VNFD Catalog

To create a VNF using the previously created we can use the following command [22]:
`tacker vnf-create --vnfd-name ud udvnf`

This command instructs Tacker to create a VNF named "udvnf" using the VNFD named "ud" that we previously onboarded to the Tacker VNFD Catalog.

Upon successful execution of the command, we should see a confirmation message indicating that the VNF has been created successfully. In the provided Fig. 12. we can observe the successful creation of the VNF.



Field	Value
created_at	2024-05-26 12:13:30.287362
description	Packet-counting
error_reason	
id	d1e348b9-ef76-423d-b14f-eb8a5414fc3a
instance_id	978b625b-0db0-4794-8818-cd9feaf88153
mgmt_ip_address	
name	udvnf
placement_attr	{"vim_name": "openstack-nfv-vim"}
status	PENDING_CREATE
tenant_id	0c52f2a0dbba46b0b66707543819f56b
updated_at	
vim_id	3ef9887f-a58c-4b3d-a045-5aea561e559c
vnfd_id	510c5b1f-f8eb-44a0-a530-23591c7d1e41

Fig 13. Successful creation of VNF

In the Instances tab, locate the VNF. Click on the name of the instance to open its detail page. In the detail page, we will find multiple tabs such as Overview, Log, Console, etc. Click on the "Console" tab. This will open a web-based console interface where we can interact with the instance as if we were connected through a direct terminal session.

```

root@ud-de96-dc7c-4e89-bc92-5cffad364792-vdu1-tncq24xttwgw:~#
root@ud-de96-dc7c-4e89-bc92-5cffad364792-vdu1-tncq24xttwgw:~# ifconfig
ens3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1442
    inet 10.10.0.1 netmask 255.255.255.0 broadcast 10.10.0.255
    inet6 fe80::f816:3eff:fed5:bdfd prefixlen 64 scopeid 0x20<link>
    ether fa:16:3e:d5:bd:fd txqueuelen 1000 (Ethernet)
    RX packets 44 bytes 6380 (6.3 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 11 bytes 1492 (1.4 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 460 bytes 33500 (33.5 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 460 bytes 33500 (33.5 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@ud-de96-dc7c-4e89-bc92-5cffad364792-vdu1-tncq24xttwgw:~# _

```

Fig 14. Result of ifconfig in VNF's console

On executing ifconfig we can see there is only one interface ens3 as in Fig. 14. so we will use the ifconfig command to bring up the additional network interfaces. Assume that we want to enable ens4, ens5, and ens6.

Execute the following commands to bring up each of the additional interfaces [34]:

```

sudo ifconfig ens4 up
sudo ifconfig ens5 up
sudo ifconfig ens6 up

```

Assign specific IP addresses and netmasks to each interface using the ifconfig command. Execute the following commands [34]:

```

sudo ifconfig ens4 10.10.1.1 netmask 255.255.255.0
sudo ifconfig ens5 10.10.2.1 netmask 255.255.255.0
sudo ifconfig ens6 10.10.3.1 netmask 255.255.255.0

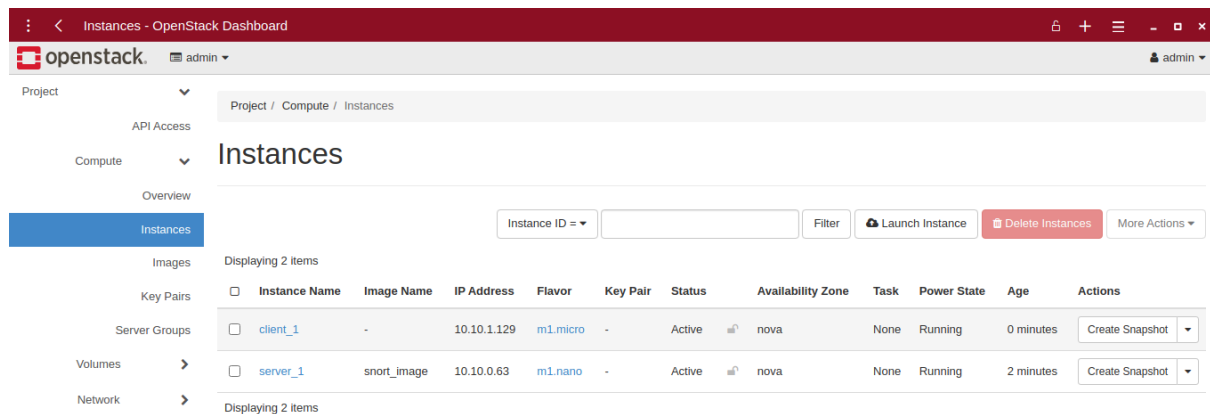
```

4.4 Configuration of Cloud Instances

To establish the architecture conducive to the deployment of a load balancer within the OpenStack environment, the creation of instances plays a pivotal role. By systematically configuring and deploying instances across different network segments, we lay the foundation for a robust and scalable infrastructure. Each instance, carefully provisioned with

the necessary resources and connectivity settings, serves a distinct purpose within the architecture. Instances labeled as Server 1, Server 2, and Server 3 are strategically positioned within network net3 to host the backend services or applications that will be load balanced. Additionally, the Health-Checker instance, situated in net2, is tasked with monitoring the health and performance of these backend servers. Meanwhile, the Load-Balancer instance, located in net1, acts as the central point for distributing incoming traffic across the backend servers, ensuring optimal resource utilization and high availability. Finally, the Client instance, residing in net0, represents the external access point through which users or external systems interact with the load-balanced services. Through the orchestrated deployment of these instances, we construct a resilient architecture capable of efficiently handling incoming requests and maintaining seamless service delivery.

Fig. 13. shows the Instances management screen. Each instance and its current state is listed, as are options to create new instances, reboot, snapshot, and terminate (delete) existing instances, and many other options.



The screenshot displays the OpenStack Horizon 'Instances' management page. The interface includes a sidebar on the left with navigation links for Project, API Access, Compute, Overview, Instances (selected), Images, Key Pairs, Server Groups, Volumes, and Network. The main content area shows a table of instances with the following data:

Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Age	Actions
client_1	-	10.10.1.129	m1.micro	-	Active	nova	None	Running	0 minutes	Create Snapshot
server_1	snort_image	10.10.0.63	m1.nano	-	Active	nova	None	Running	2 minutes	Create Snapshot

Fig 15. Instances Page in OpenStack Horizon

Launch Instance

Details

Source

Flavor *

Networks *

Network Ports

Security Groups

Key Pair

Configuration

Server Groups

Scheduler Hints

Metadata

Instance source is the template used to create an instance. You can use an image, a snapshot of an instance (image snapshot), a volume or a volume snapshot (if enabled). You can also choose to use persistent storage by creating a new volume.

Select Boot Source

Image

Create New Volume

Yes No

Volume Size (GB) *

2

Delete Volume on Instance Delete

Yes No

Allocated

Displaying 1 item

Name	Updated	Size	Type	Visibility
> CentOS 8 Stream (el8-x86_64)	11/2/21 6:49 PM	1.26 GB	QCOW2	Public

Displaying 1 item

Available 8

Select one

Q

Click here for filters or full text search.

X

Displaying 8 items

Name	Updated	Size	Type	Visibility
> Amphora (x64-haproxy-ubuntu-focal)	11/2/21 6:49 PM	359.97 MB	QCOW2	Public
> CentOS 7 (el7-x86_64)	11/2/21 6:49 PM	847.81 MB	QCOW2	Public
> CentOS 8 (el8-x86_64)	11/2/21 6:49 PM	1.22 GB	QCOW2	Public

Fig 16. Creating Instances in OpenStack Horizon

We can create a new instance by clicking Launch Instance. We will be presented with the pop-up window shown in Fig. 14.

In the Launch Instance screen, the Details tab serves as a comprehensive tool for configuring various parameters essential for instance deployment in OpenStack. Here's a breakdown of the settings available [35]:

- **Instance Name:** This denotes the name of the instance, serving as both a reference within OpenStack and the hostname of the instance.
- **Flavor:** OpenStack flavors define the virtual resource size of the instance. While DevStack creates several default flavors, their values are customizable. Set the flavor as `m1.tiny` for all the instances.
- **Instance Count:** Allows for the creation of multiple instances simultaneously by specifying the desired count. It is better to keep it 1 and create instances one by one.
- **Instance Boot Source:** Offers various options for boot sources, with "Boot from image" being a common choice, as it creates a volume based on an existing image.

- **Image Name:** Refers to the image used for instance creation, with DevStack providing at least one available image. Use the newly created ubuntu image which contains Node.js.
- **Device Size:** Specifies the size of the device associated with the instance.

Once these parameters are configured, users can proceed to the Access & Security tab to further refine access controls and security settings for the instance.

If everything works according to plan, the Status of our new instance will be Active. If something went wrong, the Status will show Error and after starting over OpenStack the status is Shut Down in which case we have to Start the instances as shown in Fig. 17. If we experience an error, we can try reducing the size of our requested virtual machine [35].

Instances

Instance ID = Filter Launch Instance Delete Instances More Actions											
Displaying 7 items											
☐	Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Age	Actions
☐	Client	blessed	10.10.3.226	m1.tiny	ssh	Shutoff	nova	None	Shut Down	1 week, 6 days	Start Instance
☐	Load-Balancer	blessed	10.10.2.208	m1.tiny	ssh	Shutoff	nova	None	Shut Down	1 week, 6 days	Start Instance
☐	Health-Checker	blessed	10.10.1.130	m1.tiny	ssh	Shutoff	nova	None	Shut Down	1 week, 6 days	Start Instance
☐	Server-3	blessed	10.10.0.22	cirros256	ssh	Shutoff	nova	None	Shut Down	1 week, 6 days	Start Instance
☐	Server-2	blessed	10.10.0.19	cirros256	ssh	Shutoff	nova	None	Shut Down	1 week, 6 days	Start Instance
☐	Server-1	blessed	10.10.0.151	cirros256	ssh	Shutoff	nova	None	Shut Down	1 week, 6 days	Start Instance
☐	ud-de96-dc7c-4e89-bc92-5cffad364792-VDU1-tncq24xtwgw	blessed	net1 10.10.1.1 net2 10.10.2.1 net0 10.10.0.1 net3 10.10.3.1	udvnt_1852de96-dc7c-4e89-bc92-5cffad364792-VDU1_flavor-ngkowsceeb5t	-	Shutoff	nova	None	Shut Down	1 week, 6 days	Start Instance

Fig 17. Successful creation of all instances

We can visualize the network topology and graph in the Networks tab. Fig. 18. and Fig. 19. likely illustrate these network connections.

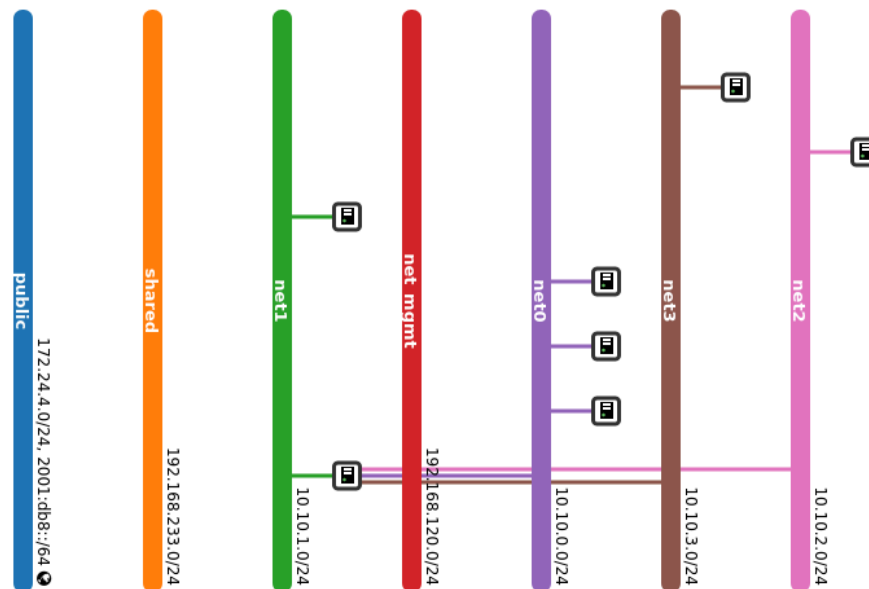


Fig 18. Network Topology of the instances

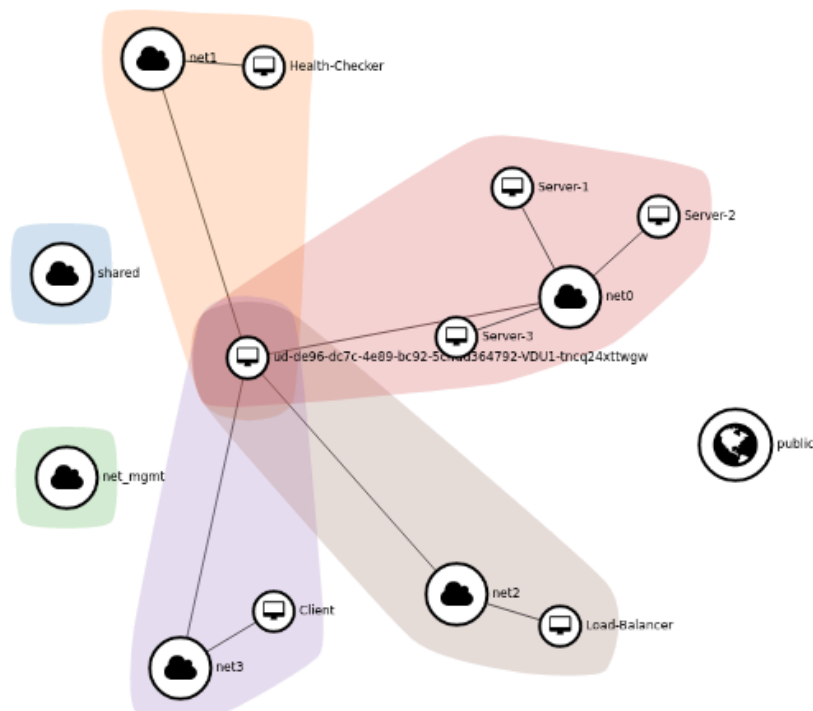


Fig 19. Network Graph of the instances

5. IMPLEMENTING LOAD BALANCER

5.1 Custom Load Balancing Strategy

To enhance the load balancing efficiency, I've replaced the conventional round-robin algorithm with a Weighted Round Robin (WRR) algorithm. Unlike the standard round-robin approach, which assigns equal weight to all servers in the pool, the WRR algorithm allocates weights to each server based on its capacity or performance metrics. This means that servers with higher capacities or better performance characteristics are assigned higher weights, allowing them to handle a larger share of incoming requests. By dynamically adjusting the weights assigned to each server, the WRR algorithm can effectively distribute the workload in a more proportional manner, optimizing resource utilization and improving overall system performance. This approach is particularly beneficial in scenarios where servers have varying capacities or where certain servers are underutilized compared to others. Integrating the WRR algorithm into the load balancing strategy offers a more intelligent and adaptive solution that can better accommodate the diverse requirements of modern applications [37].

To enhance system reliability and prevent overload situations, a proactive measure has been implemented where servers automatically stop accepting new requests once their utilization crosses a certain threshold. This threshold is defined based on predetermined performance metrics such as CPU usage, memory utilization, or network bandwidth. By temporarily halting the acceptance of new requests, servers can mitigate the risk of performance degradation or system failure caused by excessive load.

In the event that all servers become unavailable simultaneously, the load balancer seamlessly transitions into a fail-safe mode where it begins queuing incoming requests until the servers are ready to accept requests again. This queueing mechanism ensures that no requests are lost or dropped during server downtime, maintaining the continuity of service and minimizing disruptions for end users.

To ensure the continuous availability of servers and maintain optimal performance, a dedicated Health Checker Server has been deployed within the system. This Health Checker Server periodically sends requests to all servers at predefined time intervals to verify their operational status. Upon receiving a response from each server, the Health Checker Server evaluates the health and availability of the servers based on factors such as response time, status codes, and error rates.

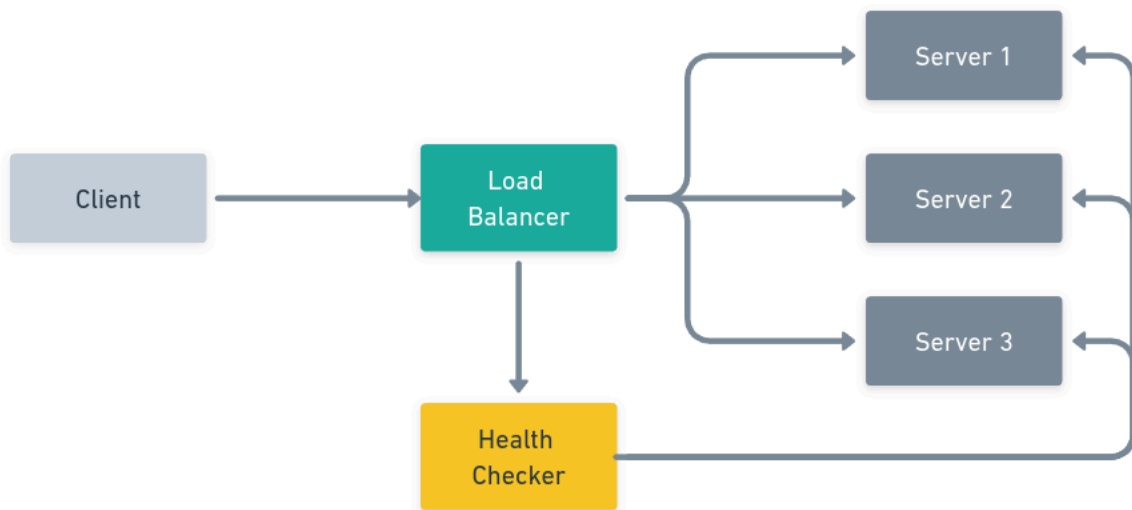


Fig. 20. Logical Architecture of Load Balancer along with Health Checker

The load balancer, in turn, regularly queries the Health Checker Server to obtain real-time updates on the current status of the servers. Fig. 15. depicts the overall architecture. By leveraging the information provided by the Health Checker Server, the load balancer can make informed decisions regarding request routing and server selection. Servers that are deemed healthy and responsive are included in the pool of available servers for request distribution, while servers that are identified as inactive or experiencing issues are temporarily excluded from the pool until they are restored to a healthy state.

This integration between the Health Checker Server and the load balancer ensures proactive monitoring and management of server health, enabling the load balancer to dynamically adapt to changing conditions and maintain optimal performance and availability for end users. Additionally, by leveraging the Health Checker Server's insights, the load balancer can implement intelligent routing strategies to optimize resource utilization and enhance overall system resilience.

5.2 Calculation using Load Balancing Metrics

During the initialization phase, the load balancer initiates communication with each server in the server pool to gather essential system metrics and performance data. This exchange of information enables the load balancer to make informed decisions regarding request distribution and server selection based on the capabilities and current workload of each server.

Upon sending a request to each server, the load balancer receives a response containing the following key metrics:

1. **Total Memory Left:** The amount of available memory (RAM) on the server, providing insight into the server's capacity to handle concurrent requests and process data efficiently.
2. **Average Clock Speed of CPU:** The average processing speed of the server's CPU, indicating the server's computational capabilities and performance efficiency.
3. **Number of CPU Cores:** The total number of central processing units (CPUs) available on the server, influencing the server's processing power and ability to handle concurrent tasks in parallel.
4. **Response Time:** The amount of time it takes for the server to process a request and send a response back to the load balancer. This metric directly impacts user experience, as longer response times can lead to slower application performance and user frustration.

In Node.js, we can utilize the `os` module to access system-related information, including total RAM, CPU speed, and number of CPUs.

By gathering these metrics during initialization, the load balancer gains valuable insights into the capabilities and health of each server in the server pool, allowing it to dynamically adjust request routing and load distribution strategies to optimize performance, enhance reliability, and ensure efficient resource utilization across the system.

To collect system metrics and response times from each server, we maintain four separate arrays within our load balancer system. These arrays store crucial data points essential for dynamic load distribution and performance optimization.

The first array, **memoryLeft**, records the available memory on each server, providing insights into their resource utilization and capacity management. This information is vital for ensuring efficient allocation of memory-intensive tasks and avoiding resource exhaustion.

The second array, **clockSpeed**, captures the average CPU speed of each server. CPU speed directly influences the processing capabilities and performance efficiency of servers, impacting their ability to handle computational tasks effectively.

The third array, **numCores**, tracks the number of CPU cores available on each server. This metric signifies the server's processing power and ability to execute multiple tasks concurrently, influencing its overall throughput and responsiveness.

Lastly, the **responseTimes** array logs the response time after making calls to each server. Response time measurement provides crucial feedback on server responsiveness and performance, enabling the load balancer to make informed decisions regarding request routing and server selection.

To normalize each array with Min-Max Normalization, we apply the following formula to each value in the array:

$$X_{normalized} = (X - X_{min}) / (X_{max} - X_{min})$$

Where:

- X is the original value in the array.
- X_{min} is the minimum value in the array.
- X_{max} is the maximum value in the array.
- $X_{normalized}$ is the normalized value of X within the range $[0, 1]$.

By applying this normalization technique, we scale each metric to a common range, facilitating fair comparison and aggregation of diverse metrics.

After normalizing each array, we obtain normalized values that range between 0 and 1, enabling consistent interpretation and integration of disparate metrics into our load balancer algorithm. This normalization process enhances the robustness and effectiveness of our load balancing strategy by ensuring that all metrics contribute proportionally to the decision-making process, regardless of their original scale or magnitude.

To handle the scenario where the normalized value becomes 0 after applying Min-Max Normalization, we adjust the value to a small non-zero value, such as 0.1, for calculation purposes. This adjustment ensures numerical stability and prevents potential issues that may arise from zero-valued inputs in subsequent calculations.

This adjustment ensures the continuity of our calculations and maintains the integrity of our load balancing algorithm, even in scenarios where the original metrics result in zero-normalized values. By incorporating this safeguard, we enhance the robustness and reliability of our load balancer system, ensuring smooth and accurate operation across various workload conditions.

We decide to prioritize response time more heavily, followed by CPU clock speed, CPU cores, and finally, memory left. We first define the weights for each metric: 5 for response time, 2 for clock speed, 2 for the number of cores, and 1 for memory left. These weights reflect the relative importance of each metric in determining server performance and suitability for handling incoming requests.

Next, we calculate the weighted score for each server using the following formula:

$$\begin{aligned} \text{Weighted Score}[i] = & 5 \times (1 - \text{Normalized Response Time}[i]) + \\ & 2 \times \text{Normalized Clock Speed}[i] + 2 \times \text{Normalized Number of Cores}[i] + \\ & 1 \times \text{Normalized Memory Left}[i] \end{aligned}$$

Where:

- i denotes the index of the server.
- $\text{Normalized Response Time}[i]$ is the normalized response time of the i th server.
- $\text{Normalized Clock Speed}[i]$ is the normalized clock speed of the i th server.
- $\text{Normalized Number of Cores}[i]$ is the normalized number of CPU cores of the i th server.
- $\text{Normalized Memory Left}[i]$ is the normalized available memory of the i th server.

By applying these weights and calculating the weighted score for each server, we prioritize servers based on their performance across multiple metrics. This weighted scoring mechanism allows us to make informed decisions when selecting servers for request distribution, ensuring optimal resource utilization and system performance within our load balancer system.

These are sample data from three servers:

Metric	Server 1	Server 2	Server 3
Response Time (seconds)	0.12	0.08	0.15
CPU Clock Speed (GHz)	2.5	3.3	2.2
Number of CPU Cores	8	4	2
Memory Left (MB)	4096	8192	2048

Table 1: Sample Data of Servers

To ensure all metrics are on a comparable scale, we apply min-max normalization to each metric array. We'll skip the mathematical steps here, but assume we follow the process explained earlier to obtain normalized values between 0 and 1 for each metric, we adjust the value to a small non-zero value, such as 0.01, for calculation purposes.

Normalized Data:

Metric	Server 1	Server 2	Server 3
Normalized Response Time	0.01	1.00	0.66
Normalized CPU Clock Speed	0.27	1.00	0.01
Normalized CPU Cores	1.00	0.66	0.01
Normalized Memory Left	0.33	1.00	0.01

Table 2: Normalized Data of Servers

Calculations:

Server 1:

$$\text{Weighted Score[Server 1]} = 5 \times (1 - 0.01) + 2 \times 0.27 + 2 \times 1.0 + 1 \times 0.33 \approx 5.80$$

Server 2:

$$\text{Weighted Score[Server 2]} = 5 \times (1 - 0.66) + 2 \times 1.0 + 2 \times 0.66 + 1 \times 0.01 \approx 7.33$$

Server 3:

$$\text{Weighted Score[Server 3]} = 5 \times (1 + 0.66) + 2 \times 0.01 + 2 \times 0.01 + 1 \times 0.01 \approx 0.03$$

Weighted Scores:

Server 1: 5.80

Server 2: 7.33

Server 3: 0.03

To establish a circular sequence of servers based on weighted scores for request handling. Organize the servers in descending order of their weighted scores to form a circular sequence by repeating the server order indefinitely. Upon receiving a new request, select the next server in the circular sequence to manage the request. If the sequence reaches the end of the server list, loop back to the beginning and continue. Additionally, to prevent consecutive requests from being routed to the same server and enhance load distribution across the system,

Randomly shuffle the circular sequence of servers. This approach ensures that servers are chosen for incoming requests based on their performance metrics and weighted scores. The circular sequencing method optimizes server resource utilization and sustains load balance throughout the system

5.3 Dynamic Server Management and Status Updates

During the initialization of the health checker, specific CPU and RAM usage limits are configured for each server. If a server's CPU or RAM usage exceeds the predefined limit, the server stops accepting new requests while continuing to process the already accepted ones. Additionally, the health checker periodically monitors the resource usage of each server and notifies the load balancer immediately upon detecting a server failure or exceeding the usage limit.

Upon receiving notification from the health checker about a server's status change, the load balancer updates its internal records to reflect the current status of the affected server. If a server is marked as unavailable due to resource exhaustion or failure, the load balancer adjusts its request routing strategy to exclude the affected server from the pool of available servers. This ensures that incoming requests are not directed to the impaired server, preventing potential performance degradation or service disruptions. This proactive approach to server management enhances system stability, resilience, and overall performance in response to changing workload conditions and server health concerns.

In the load balancing system architecture, a crucial feature is the ability to handle situations where all servers become unavailable for accepting new requests. To address this scenario, the load balancer incorporates a request queuing mechanism. When none of the servers are capable of processing incoming requests due to high resource utilization or failure, the load balancer initiates an in-memory queue to store incoming requests temporarily. This queuing mechanism ensures that incoming requests are not lost and remain queued until the servers become available again.

Upon detecting that the servers are once again capable of accepting requests, the load balancer resumes request forwarding. However, instead of immediately distributing queued requests across all available servers, the load balancer follows a specific resumption strategy. It forwards each queued request to the very next available server in the sequence that was previously established for request distribution. This approach maintains the order of request processing and preserves the fairness of request distribution among servers. This proactive approach to request management enhances system reliability, resilience, and overall performance, ultimately contributing to a seamless user experience and sustained system availability in dynamic and challenging operating environments.

Within the system architecture, the Health Checker component leverages the capabilities of the “blessed” package in Node.js to provide real-time monitoring and visualization of server metrics directly within the terminal interface. This integration empowers system administrators and operators to efficiently monitor the health and performance of servers

without the need for external monitoring tools or complex graphical interfaces. This streamlined approach to metric visualization enhances the accessibility and usability of server monitoring, enabling administrators to quickly identify and respond to potential issues or anomalies.

Moreover, the Health Checker component dynamically displays the status of each server within the terminal interface. Through color-coded indicators and textual representations, the current status of servers is prominently displayed, providing immediate visibility into the operational state of the server infrastructure.

In a similar fashion to the Health Checker, the Load Balancer component harnesses the capabilities of the “blessed” package in Node.js to enrich its functionality with detailed server monitoring and request logging directly within the terminal interface. By integrating the “blessed” package, the Load Balancer augments its operational visibility and management capabilities, empowering system administrators to effectively oversee and troubleshoot the load balancing process.

The Load Balancer provides comprehensive insights into the status and performance metrics of each server in the server pool. Through an intuitive terminal interface, administrators can access real-time updates on key metrics such as CPU usage, memory utilization, and network traffic for each server. This visual representation facilitates quick assessment of server health and aids in identifying potential bottlenecks or performance issues within the server infrastructure.

6. RESULT AND ANALYSIS

6.1 Initialising the Instances

In the Instances tab, locate the instance for which we want to open the console. This could be any of our instances like Server 1, Server 2, Server 3, Health-Checker, Load-Balancer, or Client. Click on the name of the instance to open its detail page. In the detail page, we will find multiple tabs such as Overview, Log, Console, etc. Click on the "Console" tab as in Fig. 21. This will open a web-based console interface where we can interact with the instance as if we were connected through a direct terminal session.

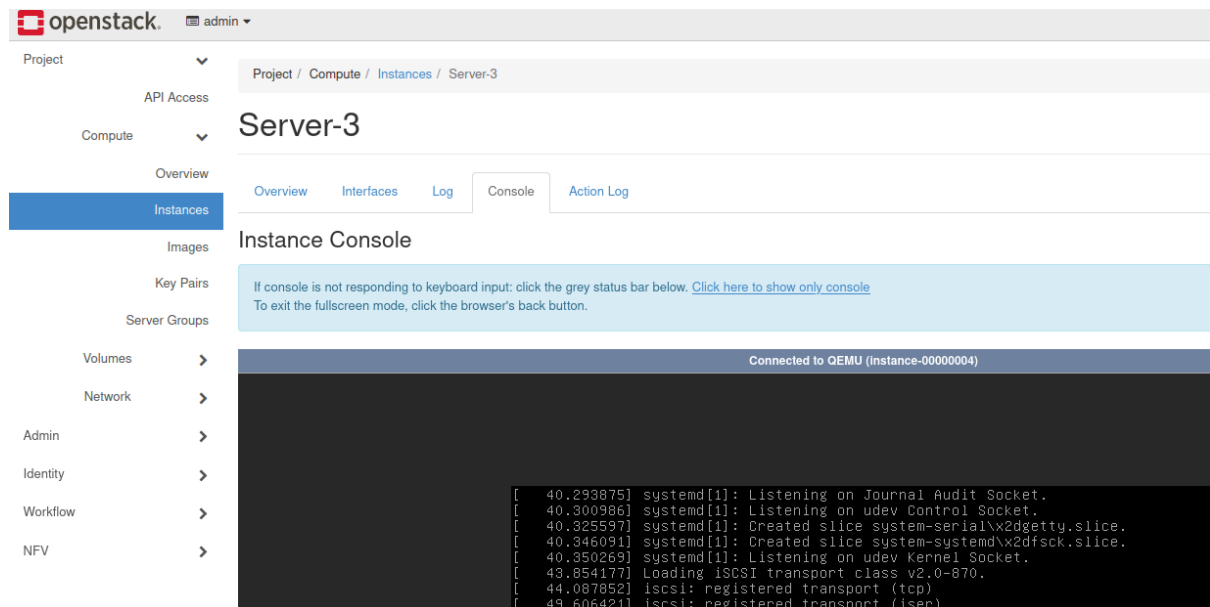


Fig. 21. Console of an Instance

When prompted, log in using the credentials we have set up for the instance as shown in Fig. 22. (typically the username and password for the operating system installed on the instance).

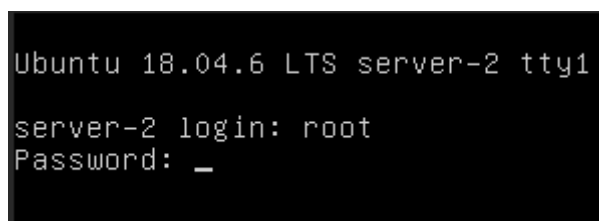


Fig. 22. Logging into an Instance

6.2 Starting the Servers

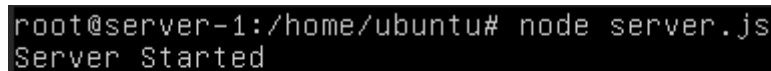
Once we are logged into the console of the instance, navigate to the directory where our server script is located.

```
cd /home/ubuntu
```

Execute the Node.js server script using the Node.js runtime. Assuming our script is named `server.js`, we can start the server with the following command:

```
node server.js
```

Repeat the above steps for each server instance (Server 1, Server 2, Server 3) to ensure all servers are running and ready to handle incoming requests. Fig. 23. shows the successful initialisation of the servers.



```
root@server-1:/home/ubuntu# node server.js
Server Started
_
```

Fig. 23. Successful initialisation of server

6.3 Initialising the Health Checker

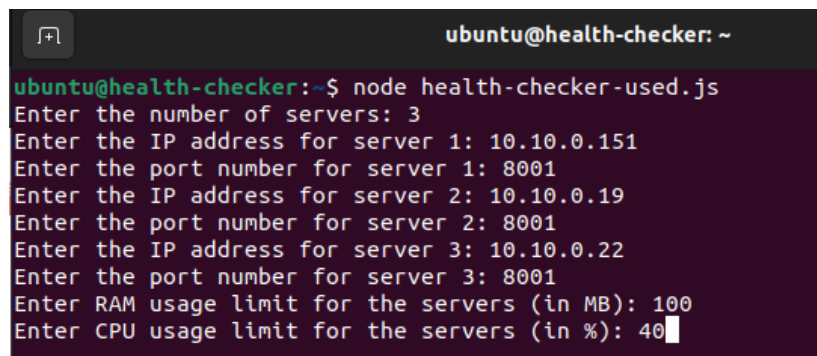
Once we are logged into the console of the Health Checker instance, navigate to the directory where our health checker script is located.

```
cd /home/ubuntu
```

Run the health checker script using Node.js. Assuming our script is named `health-checker-used.js`, we can start it with the following command:

```
node health-checker-used.js
```

When prompted, enter the number of servers we are monitoring. Enter the IP address and port for each server instance and also specify the RAM usage limit and CPU usage limit for each server as shown in Fig. 24.



```
ubuntu@health-checker: ~
ubuntu@health-checker:~$ node health-checker-used.js
Enter the number of servers: 3
Enter the IP address for server 1: 10.10.0.151
Enter the port number for server 1: 8001
Enter the IP address for server 2: 10.10.0.19
Enter the port number for server 2: 8001
Enter the IP address for server 3: 10.10.0.22
Enter the port number for server 3: 8001
Enter RAM usage limit for the servers (in MB): 100
Enter CPU usage limit for the servers (in %): 40
```

Fig. 24. Providing details of servers to health checker

Wait for a few seconds the console will start displaying the dashboard as in Fig. 25.

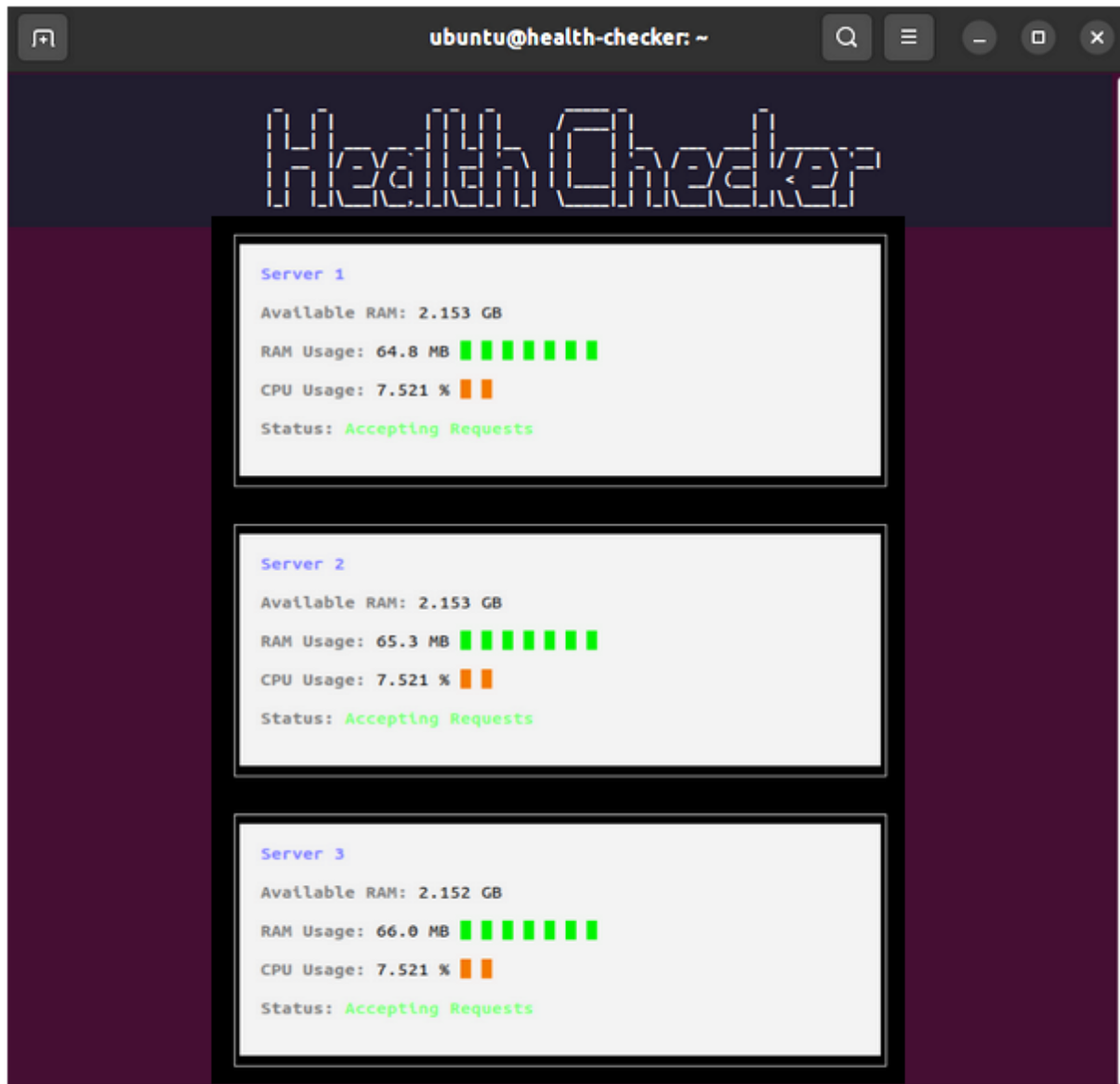


Fig. 25. Health Checker Dashboard

6.4 Initialising the Load Balancer

Once we are logged into the console of the Health Checker instance, navigate to the directory where our health checker script is located.

```
cd /home/ubuntu
```

Run the load balancer script using Node.js. Assuming our script is named load-balancer.js, we can start it with the following command:

```
node health-checker-used.js
```

When prompted, enter the number of servers we are monitoring. Enter the IP address and port for each server instance and also specify the IP address and port of the Health Checker server as shown in Fig. 26.

```

ubuntu@load-balancer: ~
ubuntu@load-balancer:~$ node load-balancer.js
Enter the number of servers: 3
Enter IP address for server 1: 10.10.0.151
Enter port number for server 1: 8001
Enter IP address for server 2: 10.10.0.19
Enter port number for server 2: 8001
Enter IP address for server 3: 10.10.0.22
Enter port number for server 3: 8001
Enter the IP address of the health checker: 10.10.1.130

```

Fig. 26. Providing details of servers to Load Balancer

Wait for a few seconds the console will start displaying the dashboard as in Fig. 27.

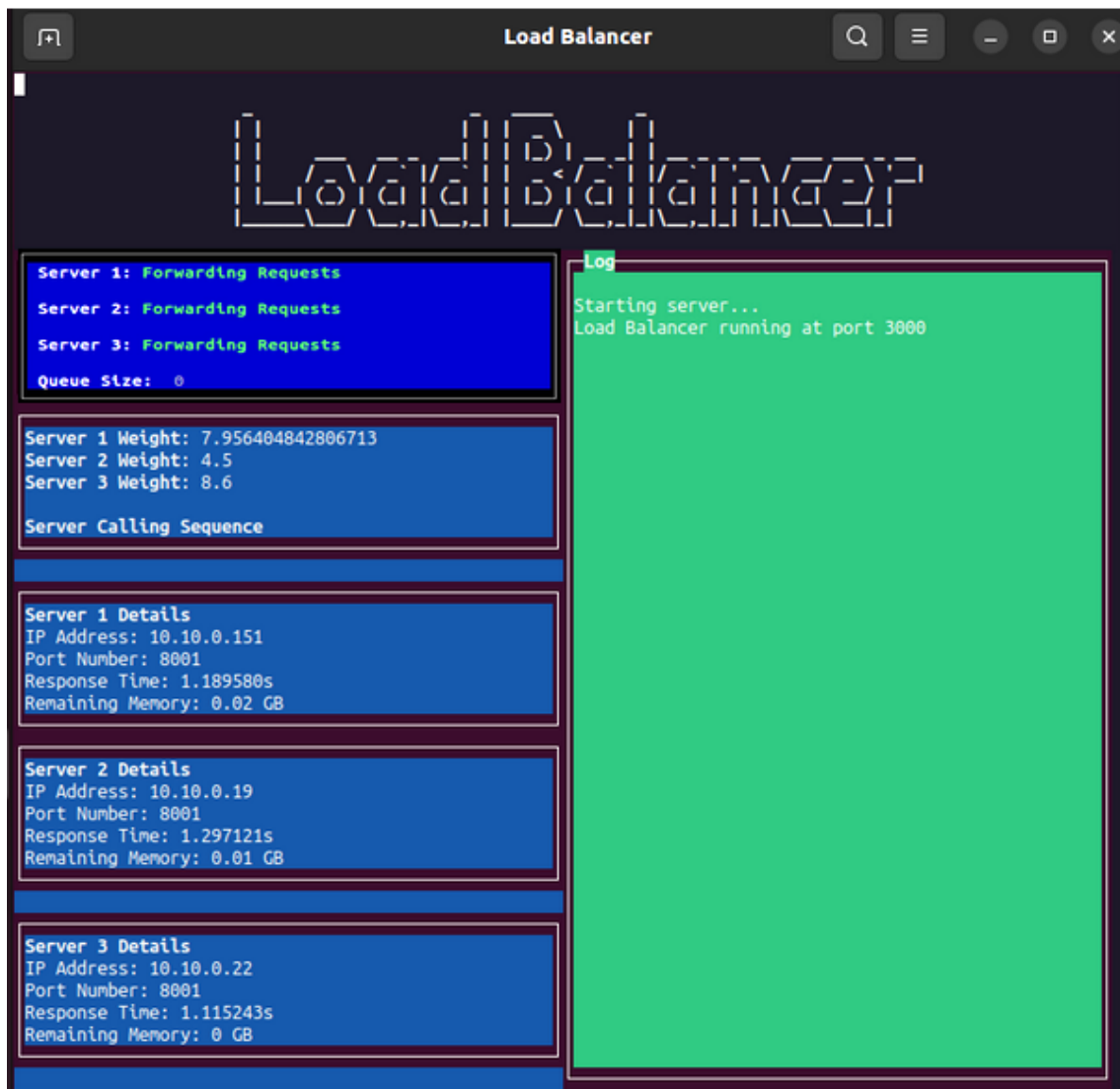


Fig. 27. Load Balancer Dashboard

6.5 Sending Requests to Load Balancer

We initiate the process of sending requests to the server using a Bash script. This script is designed to automate the transmission of HTTP requests to a designated server endpoint. Initially, it defines a function called `send_request`, which encapsulates the functionality of dispatching HTTP requests to the server using the `curl` command. This command is configured to operate silently (`-s` option), without displaying progress information, and any error messages generated are redirected to `/dev/null` to maintain script cleanliness.

```
#!/bin/bash

send_request() {
    curl http://<ip address of load balancer>:<port number>
-s 2>/dev/null &
}

read -p "Enter the number of requests to send: " num_packets

read -p "Enter the sleep duration between each packet (in
seconds): " sleep_duration

for ((i=1; i<=$num_packets; i++)); do
    send_request
    sleep $sleep_duration
done
```

The script then prompts the user to provide two parameters: the number of requests to be sent (`num_packets`) and the duration of the pause between each request (`sleep_duration`). These inputs allow for customization of the script's behavior, enabling adjustments tailored to specific testing requirements.

Subsequently, a for loop is employed to iterate from 1 to the specified number of packets (`num_packets`). Within each iteration, the `send_request` function is invoked to transmit an HTTP request to the server. Following the dispatch of each request, the script pauses execution for the specified duration (`sleep_duration`) utilizing the `sleep` command. This interval between requests simulates real-world scenarios where requests are dispatched intermittently, facilitating more accurate assessment of server performance and load-handling capabilities.

6.6 Analysis

When a client sends a request, it is directed towards the load balancer's virtual address. The load balancer employs its algorithm to select a backend server from its pool. Once a server is chosen, the load balancer forwards the client's request to the selected backend server.

In parallel with forwarding the request, the load balancer logs relevant information about the request within its own logging system as in Fig. 28. This data can be crucial for monitoring traffic patterns, identifying potential issues, and analyzing overall load balancer performance.

```
Starting server...
Load Balancer running at port 3000
Redirecting Request 0 to Server 1
Redirecting Request 1 to Server 1
Redirecting Request 2 to Server 1
Redirecting Request 3 to Server 3
Redirecting Request 4 to Server 1
Redirecting Request 5 to Server 2
Redirecting Request 6 to Server 2
Redirecting Request 7 to Server 3
Redirecting Request 8 to Server 1
Redirecting Request 9 to Server 1
```

Fig. 28. Logger in Load Balancer displaying forwarded requests

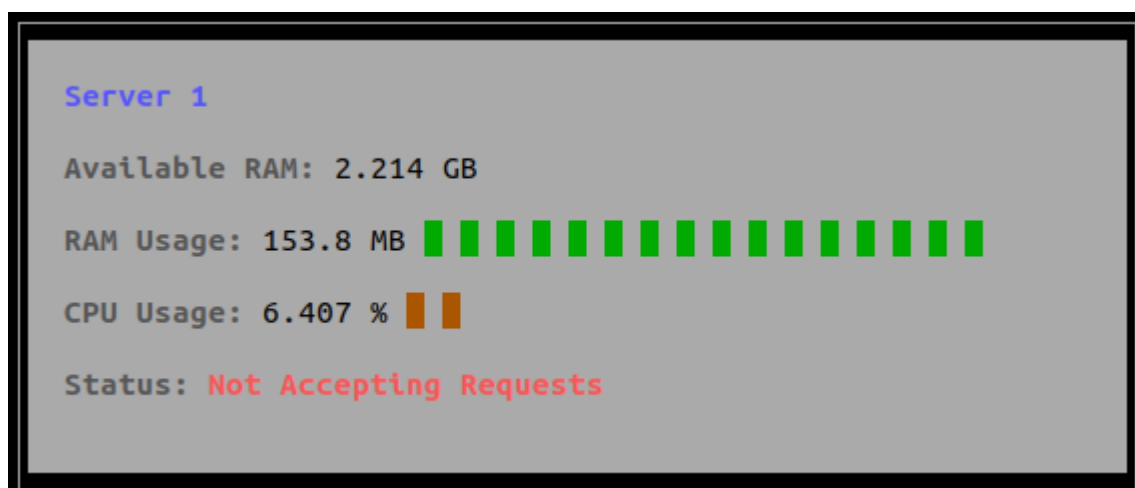
The load balancer communicates back to the client with an acknowledgement message. This message typically indicates that the request has been successfully forwarded to a backend server for processing.

Upon receiving the request, the server typically sends an acknowledgement message back to the client (through the load balancer). This message confirms that the server has received the request and is initiating the processing tasks. Once the server acknowledges the request, it begins processing the tasks associated with it. After successfully completing the tasks, the server sends a completion message back to the client (through the load balancer). The reply messages for 10 requests are shown in Fig. 29. This message indicates that the server has finished processing the request and the desired outcome is achieved.

```
Enter the number of requests to send: 10
Enter the sleep duration between each packet (in seconds): 0.2
Message from Load Balancer: Request 0 forwarded to 1
Message from Server: Request 0 accepted by Server 1
Message from Load Balancer: Request 1 forwarded to 1
Message from Server: Request 1 accepted by Server 1
Message from Load Balancer: Request 2 forwarded to 1
Message from Server: Request 2 accepted by Server 1
Message from Load Balancer: Request 3 forwarded to 3
Message from Server: Request 3 accepted by Server 3
Message from Load Balancer: Request 4 forwarded to 1
Message from Server: Request 4 accepted by Server 1
Message from Server: Operation performed successfully by Server 1 for Request 0
Message from Load Balancer: Request 5 forwarded to 2
Message from Server: Request 5 accepted by Server 2
Message from Server: Operation performed successfully by Server 1 for Request 1
Message from Load Balancer: Request 6 forwarded to 2
Message from Server: Request 6 accepted by Server 2
Message from Server: Operation performed successfully by Server 1 for Request 2
```

Fig. 29. Replies received by Client from Load Balancer and Server

The health checker continuously monitors the health of backend servers by sending requests. If a server reaches a predefined CPU or RAM usage threshold the health checker keeps a note as shown in Fig. 30.



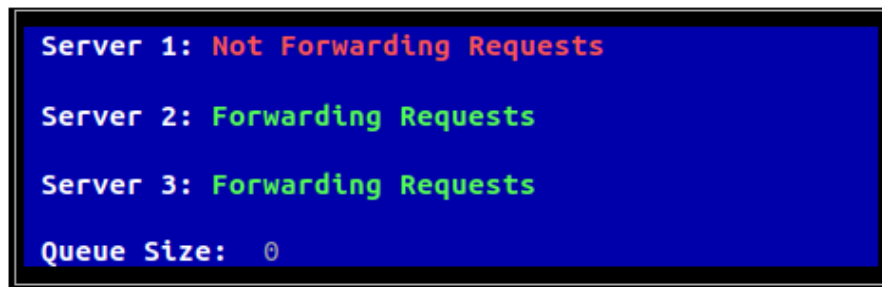


Fig. 31. Load Balancer Dashboard updating its status for unavailable server

If all backend servers become unresponsive and health checks consistently fail, the load balancer enters a fallback mode. Client requests continue to arrive, and instead of being dropped, they are stored in an in-memory queue. This queue acts as a temporary buffer, holding requests until a server becomes available again. In Fig. 32. we can see the Load Balancer keeps track of the size of the queue.

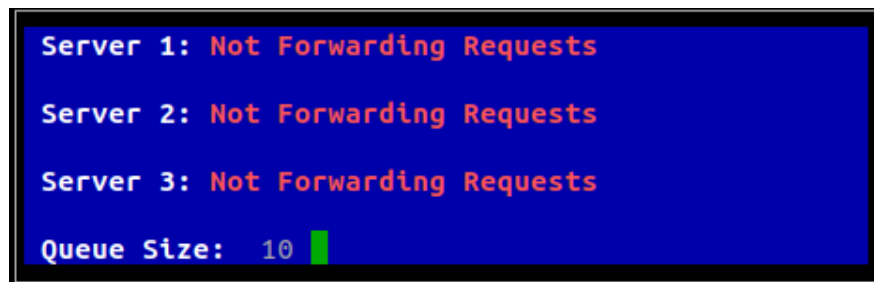


Fig. 32. Load Balancer Dashboard keeping track of size of Waiting Queue

The load balancer employs the FCFS algorithm to manage the queue. Requests are processed in the order they were received. The first request added to the queue will be the first one forwarded to a server when it becomes available. In Fig. 33. we can see the Load Balancer logs the information that the queue is currently being processed and the requests are sent to the available servers.

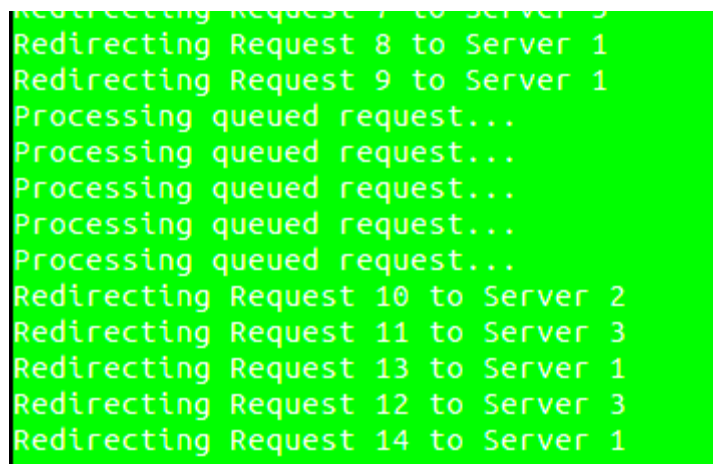


Fig. 33. Load Balancer Logger displaying message while clearing the Queue

7. CONCLUSION

The implementation of the Weighted Round Robin (WRR) load balancing algorithm in the proposed system demonstrates significant improvements in resource utilization, system performance, and resilience. Through a series of experiments and evaluations, the effectiveness and efficiency of the WRR algorithm were thoroughly analyzed, yielding valuable insights into its impact on overall system behavior and workload distribution.

In parallel with forwarding the request, the load balancer logs relevant information about the request within its own logging system. This data can be crucial for monitoring traffic patterns, identifying potential issues, and analyzing overall load balancer performance.

One of the key findings of the analysis is the ability of the WRR algorithm to dynamically adapt to changing workload conditions and server capacities. By assigning weights to servers based on their performance metrics, such as CPU speed, memory availability, and response time, the algorithm optimizes request distribution, ensuring that servers with higher capacities handle a larger share of incoming requests. This dynamic allocation of workload results in more balanced resource utilization and improved system performance, as demonstrated by reduced response times and increased throughput.

Furthermore, the proactive measures implemented to prevent server overload situations proved to be highly effective in maintaining system stability and reliability. By automatically throttling server acceptance of new requests when resource utilization exceeds predefined thresholds, the system mitigates the risk of performance degradation or failure during peak load periods. Additionally, the fail-safe mechanism of queuing requests during server unavailability ensures uninterrupted service delivery, minimizing disruptions for end users and enhancing overall user experience.

The integration of the Health Checker component provided real-time monitoring and visualization of server metrics, enabling administrators to efficiently manage and troubleshoot server health and performance issues. Similarly, the Load Balancer component displays server details and request logs, facilitating comprehensive oversight of the load balancing process. These integrated components played a crucial role in enhancing system visibility, enabling proactive monitoring, and ensuring timely response to system events.

Overall, the results of the analysis demonstrate the effectiveness of the proposed load balancing framework in optimizing resource allocation, improving system performance, and ensuring high availability and reliability in dynamic computing environments. By leveraging advanced techniques such as the WRR algorithm and proactive server management mechanisms, the system offers a robust solution for addressing the challenges of modern distributed computing and delivering seamless service to end users.

REFERENCES

- [1] *OpenStack Foundation. (n.d.). OpenStack Documentation.* Retrieved from <https://docs.openstack.org>.
- [2] *Understanding OpenStack*, Retrieved from <https://www.redhat.com/en/topics/openstack>.
- [3] Kavis, M. J. (2014). *Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS)*. John Wiley & Sons.
- [4] *OpenStack Components*. Retrieved from <https://www.openstack.org/software/project-navigator/openstack-components/>.
- [5] *OpenStack API Documentation*. Retrieved from <https://developer.openstack.org/api-ref/>.
- [6] Patel, M., & Chaudhary, S. (2016). *Advanced Load Balancing Techniques*. Springer.
- [7] White, R. (2013). *Load Balancing Servers, Firewalls, and Caches*. John Wiley & Sons.
- [8] North, S. (2019). *Practical Load Balancing: Ride the Performance Tiger*. Apress.
- [9] *Nova*, Retrieved from <https://docs.openstack.org/nova/latest/> ,
<https://docs.openstack.org/nova/latest/admin/architecture.html>
- [10] *Neutron*, Retrieved from <https://docs.openstack.org/neutron/latest/> ,
<https://docs.openstack.org/neutron/latest/install/concepts.html>
- [11] *Cinder*, Retrieved from <https://wiki.openstack.org/wiki/Cinder> ,
<https://docs.openstack.org/cinder/latest/>
- [12] *Keystone*, Retrieved from <https://docs.openstack.org/keystone/latest/>
- [13] *Glance*, Retrieved from <https://docs.openstack.org/glance/latest/>
- [14] *Swift*, Retrieved from <https://docs.openstack.org/swift/pike/admin/objectstorage-intro.html>
- [15] *Heat*, Retrieved from <https://wiki.openstack.org/wiki/Heat> ,
<https://docs.hpc.cam.ac.uk/cloud/userguide/07-orchestration.html>
- [16] *Horizon*, Retrieved from <https://docs.openstack.org/horizon/latest/> ,
https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/13/html-single/introduction_to_the_openstack_dashboard/index
- [17] *Ceilometer*, Retrieved from <https://docs.openstack.org/ceilometer/latest/>
- [18] *Trove*, Retrieved from <https://wiki.openstack.org/wiki/Trove>

- [19] *Zaqar*, Retrieved from <https://wiki.openstack.org/wiki/Zaqar>
- [20] *Ironic*, Retrieved from <https://wiki.openstack.org/wiki/Ironic>
- [21] *OpenStack Introduction*, Retrieved from <https://docs.openstack.org/contributors/common/introduction.html>
- [22] *Tacker*, Retrieved from <https://wiki.openstack.org/wiki/Tacker>,
<https://docs.openstack.org/tacker/latest/user/introduction.html>
- [23] *NFV Architectural Framework: The ETSI architectural framework explained*, Retrieved from <https://stlpartners.com/articles/network-innovation/nfv-architectural-framework/>
- [24] *virtual network functions (VNFs)*, Retrieved from <https://www.techtarget.com/searchnetworking/definition/virtual-network-functions-VNF>
- [25] *NFVi (network functions virtualization infrastructure)*, Retrieved from <https://www.techtarget.com/searchnetworking/definition/NFVi-network-functions-virtualization-infrastructure>
- [26] *NFV MANO (network functions virtualization management and orchestration)*, Retrieved from <https://www.techtarget.com/searchnetworking/definition/NFV-MANO-network-functions-virtualization-management-and-orchestration>
- [27] *DevStack*, Retrieved from <https://docs.openstack.org/devstack/latest/index.html>
- [28] *virt-customize*, Retrieved from https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_deployment_and_administration_guide/sect-guest_virtual_machine_disk_access_with_offline_tools-using_virt_customize
- [29] *npm Docs*, Retrieved from <https://docs.npmjs.com/downloading-and-installing-node-js-and-npm>
- [30] *Create and manage images*, Retrieved from <https://docs.openstack.org/horizon/queens/admin/manage-images.html>
- [31] *Create and manage networks*, Retrieved from <https://docs.openstack.org/ocata/user-guide/cli-create-and-manage-networks.html>
- [32] *VIM Management*, Retrieved from <https://docs.openstack.org/tacker/latest/cli/cli-legacy-vim.html>
- [33] *Configuration File for Registering VIM*, Retrieved from https://docs.openstack.org/tacker/latest/reference/vim_config.html

[34] *Linux ifconfig command*, Retrieved from
<https://www.computerhope.com/unix/uifconfi.htm>

[35] *How to Create an Instance in OpenStack Horizon*, Retrieved from
<https://openmetal.io/docs/manuals/operators-manual/day-1/horizon/create-first-instance>

[37] Chandra, Harish & Bahuguna, Himanshu & Sharma, Harish. (2021). *A SURVEY OF LOAD BALANCING ALGORITHMS IN CLOUD COMPUTING*.