Deployment and Operation with Automation in OpenStack Cloud

1st Madhav Sai Aryan Kothareddy
Telecommunication Systems
Blekinge Institute of Technology
Karlskrona, Sweden
maky23@student.bth.se

2nd Vamsi Bhargav Kamuju
Telecommunication Systems
Blekinge Institute of Technology
Karlskrona, Sweden
vaka20@student.bth.se

3rd C V N Krishna Vamsi
Telecommunication Systems
Blekinge Institute of Technology
Karlskrona, Sweden
vech20@student.bth.se

***Abstract*—This paper provides the software approach towards deploying network instances and configuring the nodes by using Ansible playbook in cloud using open stack and openstacksdk python module.Nova compute of openstack was used. The first section provides a brief introduction to open stack, Nova Compute and implemented solution,the solution includes three stages: Deploy, Operate and Clean up, the second section deals with statistical analysis where we implemented Apache benchmark tests and tabulated the average values of mean standard deviation and response time and in final section we performed analysis based on the results from the Apache benchmark**

***Index Terms*—OpenStack, Cloud, Nova Compute, automation, load balancing, deployment, operations, cleanup, network, components, nodes, cloud resources.**

I. Introduction

Openstack it is an opensource cloud computing platform often is deployed as infrastructure as a service (IaaS) it is highly sclable many companires are currently using this this was all made possible by the NASA and Rackspace an it web hosting company [5]

Computing Nova of openstack which is also knwon as nova compute or openstack nova.[9] which provides the management of CPU resources to the users several companies are using Nova to build their cloud. it provides infrastructure as service the compute resources provided can be accessed on demand in a scalable manner. it allows the users to create specific virtual machines with various CPU configurations. there are some fixed configurations were also available which are available in the form of flavors. the software solution discussed in this paper uses Ubuntu 20.04 Focal Fossa x86 64 and flavor:1C-2GB-50GB for its servers. [3]

Instance Configuration Manager configures the virtual machine requested by the user. The instance configuration manager has five sub components, namely, Flavor selector, Image selector, Volume, Block Storage and Network Configuration. Several flavors, i.e., predefined virtual machine configurations are available in Nova. Flavor selector selects the flavor of a virtual machine based on the configuration requested by the user. Image selector then retrieves the image of operating system requested by the user and attaches it to the virtual machine. Volume and block storage if requested by the user, are attached to the VM. The network configuration fetches the network information associated with the virtual machine and configures it. [3]

A. *Initial deployment stage*

The initial stage of solution where the script creates necessary instances in open stack cloud and generate necessary configuration files after deploying the instances in the cloud. During the creation of instances in the cloud. We utilized tags for each activity the following gives the steps followed in the initial stage.

1) **Environment setup:** The script starts by reading OpenStack credentials from a specified RC file and establishing a connection to the OpenStack environment.

2) **Key pair Generation:** The key pair will be generated in openstcak and the generated key would be stored in the default ssh directory with '.pem' extension and this stored key is utilised as identify file while connecting to remote servers.

3) **Network Setup:** The network setup includes subnet, router, and security group allowing the required ports and rules

4) **Server creation:** The script creates the necessary servers (a bastion server and two HAProxy servers). It assigns a floating IP to each server and virtual IPs were given to HAproxy servers.

5) **web server management:** The script creates the necessary web servers.

6) **Server floating IPS and Virtual IPS:** The script returns the fips of the servers and vip associated to proxy servers into files to use them later for creation of configuration files.

7) **Configuration generation:** Using the openstack commands the script fetches the internal IPS and reads the server floating IPS from a file created in previous step

and finally creates ssh configuration file and ansible host inventory file.

8) **Ansible playbook execution:** the final part of deployment stage is to install and configure the services in the remote servers for this we used anisble playbook which installs the required services in the remote host

- The script uses the OpenStack Python SDK and the OpenStack CLI to interact with the OpenStack environment. It also uses the subprocess module to execute shell commands and the json module to parse JSON output.

B. Ansible playbook

ansible playbook consists of the necessary services and installations that are to be configured in the remote servers it installs the following services

1) **Update and Install Packages**: The first set of tasks is run on all hosts. It updates the apt package cache and installs necessary packages.

2) **HAProxy ,Nginx and Keepalived Configuration**: The next set of tasks is run on the main proxy and standby proxy hosts. It installs HAProxy and Keepalived, Ngnix configures them using templates and starts the services.

   We have a PROXY node running NGinx and HAproxy, which we have built to achieve load balancing for the service. The steps required to set up and configure the PROXY nodes for load balancing traffic amongst the services are included in the Ansible playbook (site.yaml).

   On the service nodes, SNMPd and Python are operating. Two PROXY nodes are deployed, and keepalived is used to ensure redundancy and remove Single-Point-Failure (SPF) issues.

   The two PROXY nodes can share a floating IP thanks to the keepalived configuration, which facilitates smooth fail over in the event of a node loss. In the event that one PROXY node fails, the other node assumes control of the floating IP, guaranteeing continuous service availability and access. [2]

3) **Web Server Configuration:** The next set of tasks is run on the web servers. It installs necessary packages, creates a directory, copies a Python script to the servers, and starts a Flask app.

4) **Grafana and Prometheus**: The next set of tasks is run on the bastion host. It installs Grafana and Prometheus, configures them, and starts the services. After it is launched, Prometheus is in charge of gathering metrics that are made available by different parts, such as the Node Exporter that is installed on every service node. These metrics are kept by Prometheus in a timeseries database, allowing for performance tracking and historical analysis. [1] Grafana and Prometheus work together to create a dashboard that is easy to use and customise for data visualisation and alerting. Grafana enables users to construct interactive charts, graphs, and dashboards for real-time monitoring of system performance, resource utilisation, and service health. [1]

5) **Node Exporter Installation**: The final set of tasks is run on all hosts. It downloads and installs Node Exporter, a Prometheus exporter for hardware and OS metrics, and starts the service. with the node exporter we get valuable information of the node performance and health. [1]

The playbook uses several Ansible modules, including apt for package management, template for managing files, service for managing services, and shell for running shell commands. The playbook uses Jinja2 templates for generating configuration files. Jinja2 is a modern and designer-friendly templating language for Python. Haproxy was used as it is simple to configure for the load balancing we implemented roundrobin algorithm which is a static based algorithm and it is one of the best load balancing algorithm. [4].

Nginx offers advantages like low consumption high concurrency it is used in small medium cloud platform as load balancing component.it is configured to handle HTTP and UDP traffic, providing a flexible and high-performance web server and reverse proxy solution. The configuration includes a health check endpoint to monitor server status, ensuring that only healthy servers receive traffic. [2]

C. Operations mode

In this stage the script monitors continuously and manages the environment, the monitoring is described as follows

1) Reads the required number of development servers from a file named servers.conf.
2) Gets the list of existing servers using the openstack server list command.
3) Fetches network parameters from OpenStack based on the naming convention.
4) Manages the development servers based on the required number of servers. If there are fewer servers than required, it creates the necessary servers. If there are more servers than required, it deletes the excess servers.
5) Updates SSH and Ansible host inventory files.
6) Waits for 30 seconds to allow the servers to start up.
7) Runs the Ansible playbook for further configuration of the newly added servers.
8) If the required no of servers meet the script sleeps for 30 seconds and checks again

D. *Clean up*

In this stage the script releases the instances in the environment and deletes the configuration files created during the initial deployment stage ensuring every thing is released and environment is empty

E. *Motivation*

The design utilizes the open stack cloud and automation of ansible for deployment operation and cleanup the design has several benefits this making it an good choice for managing the network environment.

- Consistency and Simplicity: The consistent and repetitive setup and teardown of the network infrastructure are guaranteed by the use of scripts for deployment and cleanup. Automation minimises configuration discrepancies and reduces the likelihood of human errors.

- Scalability: Continuously monitoring the number of active development servers, the operation stage script automatically adjusts their count in accordance with the servers.conf file. This scalability feature enables the network to accommodate a variety of demands without the need for manual intervention.

- Maintainability and Flexibility: The modular design, which enables simpler maintenance and updates, is achieved by managing each stage with separate scripts. It is possible to make modifications to specific phases without affecting the entire process.

- Efficiency: The utilisation of Ansible for configuration management improves efficiency. The idempotent nature of Ansible guarantees that configuration tasks are executed only when they are required, thereby minimising superfluous modifications and enhancing the stability of the environment.

- Reusability: The cleansing stage script guarantees that all resources are eliminated, thereby preparing the environment for reuse. This is especially advantageous when the network infrastructure is transient or necessitates frequent reconstruction.

- Infrastructure as Code (IaC): The concept of IaC is consistent with the utilisation of scripts for deployment, operation, and cleansing. Version control, collaboration, and documentation are all facilitated by managing the network as code.

F. *Alternatives considered*

Although the chosen design offers significant benefits, it's likely that other network management strategies were also explored. Some possible alternatives might include:

- **Manual Configuration:** Setting up the network infrastructure by hand is a common practice, especially in smaller setups. However, this approach can lead to errors, take a lot of time, and may be difficult to replicate with consistency.
- **Other Configuration Management Tools:** Instead of Ansible, the team might have considered other configuration management tools like Puppet, Chef, or SaltStack. Each of these tools has its unique features, and the selection might depend on the team's familiarity or specific project needs.
- **Containerization:** Implementing container-based solutions with tools like Docker or Kubernetes could be another option for managing the network's application components. This strategy offers advantages like isolation, portability, and scalability for the services running within the network.
- **Infrastructure Automation:** Using tools like Terraform for provisioning infrastructure resources could provide a more declarative approach to managing infrastructure. Terraform can be paired with configuration management tools to deliver a comprehensive solution.
- **Cloud-Native Management:** In cloud environments, using native services such as AWS CloudFormation or Azure Resource Manager might be considered. These tools offer deep integration and automation capabilities tailored specifically for cloud infrastructer.
- **Configuration Templating:** Rather than creating configuration files directly, using templating engines like Jinja2 or Mustache could be another option. Templating allows for more dynamic and adaptable configuration generation.

In summary, the selected approach—utilizing scripts for deployment, operation, and cleanup, along with Ansible for managing configurations—provides a solid and effective method for automating network infrastructure management. The design is scalable, easy to maintain, and well-suited for networks of varying scales.

II. Statistical analysis

In this section we performed Apache benchmark test on the nodes to understand how the nodes performance of the solution and the service deployed and how it is effected by the no of nodes used. The test was conducted in several stages varying the no of requests and concurrency level and varying the no of nodes,the following tables are the result of performance tests done on nodes the test was performed from the local machine by sending requests to HA proxy public ip. the data was collected from the connection times section of the test output.

Node	Mean (msec)	Standard Deviation (msec)	Response Time (msec)
1	89	2.9	89
2	84	1.4	84
3	91	3.0	93
4	85	1.3	85
5	105	6.3	108

TABLE I
AVERAGE VALUES OF THE MEAN, STANDARD DEVIATION, AND RESPONSE TIME OF CONNECTION TIME WITH N=10, C=10

Node	Mean (msec)	Standard Deviation (msec)	Response Time (msec)
1	126	26.5	125
2	134	11.1	134
3	127	15.8	131
4	126	15.0	132
5	113	20.4	120

TABLE II
VALUES OF MEAN, STANDARD DEVIATION, AND RESPONSE TIME OF CONNECTION TIMES WITH N=100, C=100

Node	Mean (msec)	Standard Deviation (msec)	Response Time (msec)
1	425	106.3	408
2	433	310.6	320
3	532	284.4	428
4	556	328.7	461
5	547	286.0	492

TABLE III
VALUES OF MEAN, STANDARD DEVIATION, AND RESPONSE TIME OF CONNECTION TIMES WITH N=1000, C=1000

when the benchmark test was conducted for larger value it resulted as shown in following figure indicating that the operating system's limit on the number of open files or sockets has been exceeded. This issue can occur when Apache Bench (ab) is trying to open a large number of connections simultaneously, and the system's configuration restricts the maximum number of open files or sockets that a process can have.

```
(.venv) madhav@RADHA-PC:~/NSO_Project103$ ab -n 10000 -c 10000 http://89.46.80.144:5000/
This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 89.46.80.144 (be patient)
socket: Too many open files (24)
```

Fig. 1. N=10000 C=10000

III. ANALYSIS AND ARCHITECTURAL CONSIDERATIONS

In this section through statistical approach we evaluated our solution's operation on large scale scenarios both in service and network perceptive, according to the benchmark test result for n=1 and c=1 the requests per second value was 11.57 milliseconds.based on this we made some estimated calculations for different no of users

- Average Service Time for the test case with n=1 c=1 is: 11.57 m/sec
- Requests per Second the number of requests one server can handle per second is:

$$\begin{aligned}\text{Requests per second} &= \frac{1}{\text{Average service time}} \\ &= \frac{1}{0.044} \\ &\approx 22.73 \,\text{requests/second}\end{aligned}$$

Number of requests one server can handle per second = requests/second

- Number of Servers Required For different user loads, the number of servers required can be calculated as follows: For 100 users/second:

$$\begin{aligned}\text{Number of servers} &= \frac{100 \,\text{users/second}}{22.73 \,\text{requests/second}} \\ &\approx 4.39 \\ &\approx 5 \,\text{servers}\end{aligned}$$

For 1000 users/second:

$$\begin{aligned}\text{Number of servers} &= \frac{1000 \,\text{users/second}}{22.73 \,\text{requests/second}} \\ &\approx 43.99 \\ &\approx 44 \,\text{servers}\end{aligned}$$

For 10000 users/second:

$$\begin{aligned}\text{Number of servers} &= \frac{10000 \,\text{users/second}}{22.73 \,\text{requests/second}} \\ &\approx 439.9 \\ &\approx 440 \,\text{servers}\end{aligned}$$

- Average service time: 0.044 seconds
- Requests per second: 22.73 requests/second
- Number of requests one server can handle per second: 22.73 requests/second

- Number of servers required:
- For 100 users/second: 5 servers
- For 1000 users/second: 44 servers
- For 10000 users/second: 440 servers

based on the calculations made we can conclude that the to match and handle more no of users we need a better solution and alternatives are to be considered.

Impacts on service performance and problems

- **Resource Exhaustion**: As the number of users increases, the resources available on each individual node may become depleted. resulting in decreased performance and the possibility of service interruption. The significance of this problem on service performance is crucial, as it directly influences user experience and service availability.

- **Network delay**: When working from different locations, network delay might be a big issue. Elevated latency

can result in diminished response times and have a detrimental effect on the overall user experience.

- **Data Consistency**: In a distributed architecture that extends across numerous locations, ensuring data consistency among nodes becomes a difficult task. Inconsistencies can result in data integrity problems, which can affect the precision of user interactions.
- **Managing the service from several sites presents various difficulties**:
- **Load balancing**: it is the act of efficiently distributing user traffic over several sites, taking into account factors such as network delay and server load, in order to achieve optimal performance. Network Security: Ensuring the safety of data while it is being sent between different locations necessitates strong encryption and security protocols.
- **Latency**: The distance between locations might cause latency, which may lead to delayed reaction times.
- **Data Synchronisation**: Ensuring the consistency of data among dispersed nodes gets intricate. Implementing synchronisation techniques is crucial for ensuring accurate and current information across the server.

REFERENCES

[1] Lei Chen, Ming Xian, and Jian Liu. Monitoring System of OpenStack Cloud Platform Based on Prometheus. In *2020 International Conference on Computer Vision, Image and Deep Learning (CVIDL)*, pages 206–209, July 2020. URL: https://ieeexplore-ieee-org.miman.bib.bth.se/document/9270544/?arnumber=9270544, doi:10.1109/CVIDL51233.2020.0-100.

[2] Min Han, D. G. Yao, and X. L. Yu. A Solution for Instant Response of Cloud Platform Based on Nginx+ Keepalived. In *Proceedings of the International Conference on Computer Science, Communications and Multimedia Engineering, Beijing, China*, pages 24–25, 2019. URL: https://scholar.archive.org/work/3kzcidurtngu5flq74kxcmwz5i/access/wayback/http://dpi-proceedings.com/index.php/dtcse/article/download/32516/31106.

[3] Pragya Jain, Aparna Datt, Anita Goel, and S. C. Gupta. Cloud service orchestration based architecture of OpenStack Nova and Swift. In *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 2453–2459, September 2016. URL: https://ieeexplore-ieee-org.miman.bib.bth.se/document/7732425/?arnumber=7732425, doi:10.1109/ICACCI.2016.7732425.

[4] Fatma Mbarek and Volodymyr Mosorov. Load balancing algorithms in heterogeneous web cluster. In *2018 International Interdisciplinary PhD Workshop (IIPhDW)*, pages 205–208, May 2018. URL: https://ieeexplore-ieee-org.miman.bib.bth.se/document/8388358, doi:10.1109/IIPHDW.2018.8388358.

[5] Tiago Rosado and Jorge Bernardino. An overview of openstack architecture. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, IDEAS '14, pages 366–367, New York, NY, USA, July 2014. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/2628194.2628195, doi:10.1145/2628194.2628195.