Service Deployment and Operation in an Openstack cloud

https://github.com/AkhilDasyam/NSOproject.git

1st Venkat Sai Akhil Dasyam
Dept. of Telecommunication Systems
Blekinge Institute of Technology
Karlskrona, Sweden.
veda21@student.bth.se

2nd Chowki Manichandana
Dept. of Telecommunication Systems
Blekinge Institute of Technology
Karlskrona, Sweden
macn21@student.bth.se

3rd Sai Praneeth Reddy Ganta
Dept. of Telecommunication Systems
Blekinge Institute of Technology
Karlskrona, Sweden
saga22@student.bth.se

***Abstract*—This project presents one of the approaches to establishing and maintaining a robust and dependable service within an OpenStack Cloud environment. Its core objective is to ensure the service remains consistently prepared to handle substantial user loads and operate seamlessly. To achieve this, the project employs OpenStack for efficient management of technical elements such as network configuration. Additionally, monitoring tools like Grafana and Prometheus are implemented to meticulously track system performance. Load distribution for incoming web traffic is achieved through the integration of HAproxy and NGinx. Furthermore, the incorporation of Keepalived enhances system reliability by ensuring continuous operation even in the face of potential disruptions. The project is structured into three primary phases: initialization, operation, and orderly shutdown. Oversight of the entire process is facilitated by a dedicated Bastion system, which monitors and optimizes service performance. Collectively, these steps exemplify a sophisticated and well-coordinated strategy for optimizing service functionality within an OpenStack Cloud environment.**

***Index Terms*—openstack, ansible, automation, prometheus, Nginx, HAproxy, SNMP, grafana.**

A. Github repository

https://github.com/AkhilDasyam/NSO_project.git

I. Introduction & Design

Our software solution includes creating a system that uses an openrc file with the required login information to access a particular OpenStack Cloud. The solution uses a special TAG during deployment to enable quick identification and tracking of every object created within the OpenStack Cloud.

This project delves into the development of a comprehensive software solution designed to address these challenges by orchestrating a streamlined process for deploying, operating, and optimizing services within an OpenStack Cloud framework. OpenStack, renowned for its robust infrastructure management capabilities, forms the foundational backbone of this endeavor. The project's central focus lies in ensuring the reliability, scalability, and availability of the deployed services, all while making use of various automation techniques.

To access the cloud, a file (openrc) is used for authentication of the openstack user. Then, all the features of openstack are accessible to the user and can carry out the functions like creation or deletion of any network, server or any other elements easily.

The project unfolds in three distinct phases: Installation, Operation, and Cleanup.

1) Install Mode: Execute the install.sh script. This uses OpenStack commands to create foundational network infrastructure components i.e., networks are deployed, routers are configured, nodes are provisioned, dependencies are managed, and services are deployed. This phase not only sets the stage for the subsequent operational activities but also showcases the practical application of network design principles and system configuration.

The install.sh script generates an SSH config file that contains the necessary attributes for SSH connections to the created servers. It also generates a hosts file that should be used in conjunction with the ansible script (site.yaml). Configuration using site.yaml:

After creating the network elements and generating the necessary files, the project utilizes the site.yaml file in the Ansible script. The site.yaml file contains tasks for configuring and managing various components of the infrastructure.

2) Operate Mode: This phase embodies the essence of dynamic scalability and continuous monitoring. The solution diligently monitors node availability and autonomously manages node count, adeptly responding to evolving user demands. Previously installed services like load balancing, a critical aspect of ensuring equitable resource allocation, is achieved through the strategic deployment of HAproxy and NGinx on proxy nodes, bolstered by the fault-tolerant Keepalived system by handling all the new changes in operational phase. Execute the operate.sh script. This reads the desired number of nodes from the servers.conf file. The operate.sh script adjusts the number of servers by either creating new servers or deleting existing ones using OpenStack commands. The adjustments ensure that the infrastructure matches the desired state, allowing for scaling up or down based on the information

provided in servers.conf

3) Cleanup Mode: This phase is often overlooked but equally pivotal, is diligently executed to release and remove cloud resources allocated during the deployment and operational phases. This phase embodies prudent resource management, preventing wastage and maintaining an organized cloud environment.

Execute the clean-up script. This utilizes OpenStack commands to delete all resources created during the installation and operation phases. This includes deleting servers, subnets, networks, key pairs, ports, and any other network elements created in the cloud platform. The clean-up mode ensures a clean state by removing all infrastructure components.

Each mode of operation—Install, Operate, and Cleanup—is meticulously orchestrated through dedicated scripts, seamlessly integrating OpenStack commands, and aligning with objectives.

A. Motivation

Our design incorporates the use of Ansible scripts for deploying services such as HAProxy, Flask services, and SNMP services. Additionally, we utilize shell scripts for executing OpenStack commands and creating configuration files. This choice of tools and approach offers significant advantages in terms of troubleshooting and ease of maintenance.

By using Ansible for service deployment, we benefit from its robust error handling and logging capabilities. If any errors occur during the deployment process, Ansible provides clear error messages that help pinpoint the root cause. This makes troubleshooting much easier and more efficient, as we can quickly identify the specific task or module that encountered the issue.

We are utilizing shell scripts to execute OpenStack commands and create configuration files offers a dependable and well-known approach for troubleshooting. Shell scripts yield comprehensive output, enabling the detection of errors or problems during their execution. Furthermore, incorporating logging mechanisms within these scripts allows the capture of pertinent details to aid in troubleshooting efforts.

In our straightforward architecture, we employed Prometheus as our monitoring solution for overseeing all servers, while Grafana served as the visualization tool for presenting the metrics under observation.

Our design seems to strike a balance between automation, efficiency, and scalability, The chosen approach aligns well with the complexities of managing an OpenStack Cloud infrastructure while providing the opportunity for users to understand the intricacies of cloud deployment and management.

B. Alternatives

As mentioned earlier we use shell scripts for automating the cloud infrastructure process (i.e., network, servers, creation and manipulation). So, instead of shell scripts, we could use terraform, python etc which are more powerful than shell scripting. Terraform offers more cloud infrastructure management capability and provides better version control and Python offers more API interaction and easier codes.

In monitoring, we used prometheus and grafana. Instead of prometheus, we can use other tools like InfluxDB which offers high performance and scalability for handling large volumes of time-series data.

II. Performance Analysis of the System

This investigation aims to analyze how the system's performance (i.e., connection times) is affected if the number of nodes are varied. To evaluate the request performance, we used Apache Benchmark (ab) tool. Our focus is on the mean and standard deviation values in the 'Connection Times' section of the benchmark results. So, in our setup, there are 2 variables i.e., connection times (dependent variable) and number of nodes (independent variable).

A. Experimental Setup

The experiments were conducted using the Cleura Cloud platform. The same system setup which was mentioned in the Introduction section was used. The setup contains one Bastion Server, 2 HAproxy servers (one main and the other is backup) and the number of web server nodes are varied from 1 to 5 in this experiment. The apache benchmark tool [3] is used on the IP address of the Proxy nodes. For every varying scenario (changing nodes), the number of requests and the concurrent connections are kept constant.

B. Procedure & Data collection

The experiments were performed for 1, 2, 3, 4, 5, and 6 nodes i.e., the range of this variable is 1 to 6, with each case tested individually. For each test, 1000 requests were sent concurrently with a concurrency level of 100. The benchmarking was repeated multiple times to ensure the consistency and reliability of the results. The recorded mean and standard deviation values for the 'Connection Times' section were noted for analysis.

To ensure statistical significance, we performed multiple iterations of the benchmarking process and collected data from each run. The output for apache benchmark tool for each case was displayed in Figure 2. The mean and standard deviation values were obtained based on the aggregated results and arranged in the following table I

Nodes	Connect		Processing		Waiting		Total	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
1	25	2.9	42	10.1	42	10.1	67	11.3
2	31	6.1	38	8.9	38	8.9	69	11.9
3	26	2.3	29	4.5	29	4.5	54	5.6
4	41	16.6	39	12.4	39	12.4	80	19.9
5	31	4.7	34	8.3	34	8.3	64	10.0
6	27	3.2	29	8.9	28	3.9	56	9.8

TABLE I
APACHE BENCHMARK TOOL RESULTS

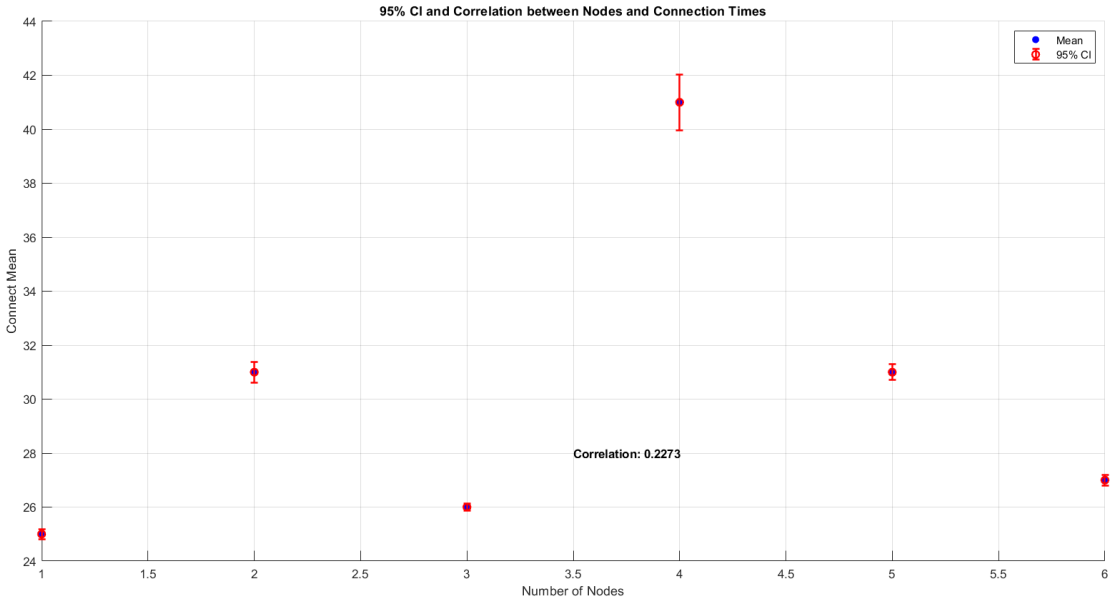

Fig. 1. Visualization

```
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Completed 1000 requests
Finished 1000 requests


Server Software:        gunicorn
Server Hostname:        188.212.109.39
Server Port:            5000

Document Path:          /
Document Length:        63 bytes

Concurrency Level:      100
Time taken for tests:   0.808 seconds
Complete requests:      1000
Failed requests:        110
   (Connect: 0, Receive: 0, Length: 110, Exceptions: 0)
Total transferred:      196912 bytes
HTML transferred:       62912 bytes
Requests per second:    1237.14 [#/sec] (mean)
Time per request:       80.832 [ms] (mean)
Time per request:       0.808 [ms] (mean, across all concurrent requests)
Transfer rate:          237.90 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       26   32   2.6     32      41
Processing:    29   38   6.6     37      73
Waiting:       29   38   6.6     36      72
Total:         56   71   7.9     69     110

Percentage of the requests served within a certain time (ms)
  50%     69
  66%     71
  75%     73
  80%     74
  90%     78
  95%     86
  98%     97
  99%    105
 100%    110 (longest request)
```

Fig. 2. Apache benchmarking command execution

C. Statistical Analysis

After executing the apache command for each scenario, we obtained the outputs just as shown in Figure 1. So according to that, we extracted the mean and standard deviation data of connect, processing, waiting and finally the Total times for each node case (1 to 6), the extracted data from the above tool is given in TABLE I

We took the help of MATLAB for the calculation, visualization, and plotting of the obtained data, first, we obtained the 95 Confidence interval for each node cases connect times mean value. And after that, we found out the correlation value between number of nodes and connection times which is found out to be **0.2273**. And to visualize the collected and calculated data and understand the pattern, we plotted the graph mean values of connection times with a 95 confidence interval.

In MATLAB, to findout 95CI we used the formula :

$$\text{CI} = \bar{x} \pm Z \left(\frac{s}{\sqrt{n}} \right)$$

Where:
CI = Confidence Interval
$\bar{x}$ = Sample Mean (obtained from ab tool)
Z = Z-score for the desired confidence level (here, 1.96 for 95 confidence)
s = Sample Standard Deviation (obtained from ab tool)
n = Sample Size (here total number of requests are 1000 for every case)

To ensure a more deeper relationship between the nodes and connect times of the requests, we performed ANOVA test (Analysis of variance test) . We assumed the hypotheses as:

Null hypothesis: There is no significant difference in connect times while varying nodes.

Alternate hypothesis: There are significant differences in connect times while varying nodes.

We conducted the ANOVA test in MATLAB using function 'anova1' from Statistics and Machine learning toolbox. We considered the significance level (alpha) as 0.05. After performing the test, we got p value as 0.5157.

D. Conclusion

By observing the values from TABLE I we can state that as the number of nodes increases from 1 to 6, the "Connect" time mean shows some variation. The "Connect" time mean is relatively consistent for 1, 3, 5, and 6 nodes, with minor fluctuations. There is a relatively higher "Connect" time mean for 2 and 4 nodes compared to the other configurations.

By seeing the pattern in Figure 1, firstly, we can state that, the confidence intervals for data of each node case are mostly narrow. We already know that more narrow CIs means more precise estimates. And we can see that the mean values are generally tending to increase in a short change of number of nodes but by viewing the whole change in the spectrum from 1 to 6, the change in the mean of connect times is nonlinear. For instance, the mean. This indicates that the relationship between the mean value and the number of nodes is not strictly linear.

By the correlation value i.e., 0.2273 we can say the number of nodes and connect times has a positive correlation but it has a significantly less magnitude. This indicates that as the number of nodes increases the connection times increase, but the linearity is significantly weak. From the ANOVA test, we can see that the obtained, p value is greater than alpha, so we failed to reject null hypothesis i.e., null hypothesis is true. By this, we can say that observed differences in connect times could reasonably be explain by randomness.

So, we can conclude that the variables have a slight tendency to move in the same direction but it does not necessarily indicate a cause-and-effect relationship between them. We also have to remember many factors that could potentially influence the connection times at that particular instance of measurement. Many factors like network congestion, resource allocation, and system architecture can contribute to these irregularities in results.

III. Scalability and Future Growth Considerations

A) Network Perspective

Bandwidth: Considering our service would be a hit, it is safe to assume that a single user's action would be capable of transferring data of at least 10 Mega bits (e.g., 10 Mb). Therefore, for each number of users case, the bandwidth calculation is as follows:

$$100\,\text{users/s} = 100 \times 10\,\text{Mb} = 1000\,\text{Mb/s} = 1\,\text{Gbps}$$
$$1000\,\text{users/s} = 1000 \times 10\,\text{Mb} = 10000\,\text{Mb/s} = 10\,\text{Gbps}$$
$$10000\,\text{users/s} = 10000 \times 10\,\text{Mb} = 100000\,\text{Mb/s} = 100\,\text{Gbps}$$

Network Architecture: With the expectation of our service becoming a major hit, network architecture upgrades could involve:

- Strategic deployment of servers in multiple regions or data centers to reduce latency and distribute load.
- Utilization of newer technologies like edge or fog computing to bring computing resources closer to users or distribute them over multiple levels.
- Implementation of improved queuing algorithms to evenly distribute load and support a larger user base.
- Adoption of asynchronous processing of user requests to enhance load handling capabilities.

Load Balancing: As our user base grows, load balancing upgrades could include:

- Adoption of multi-layered load balancing techniques, such as DNS-based load balancing, network-level balancing, and application-level balancing.
- Implementation of dynamic load balancing algorithms based on CPU load, memory utilization, and network latency, assigning server weights accordingly.

B) Service Perspective

Server Requirements: Estimation of computational resources (CPU, memory, storage) required per user action and user, and calculation of the total resources needed for different user levels.

Database Scalability: Evaluation of database system scalability, considering sharding, replication, or NoSQL databases to handle increased data storage and access demands.

Caching: Implementation of caching mechanisms (e.g., in-memory caching) to reduce backend load and enhance response times.

Microservices Architecture: Consideration of a microservices architecture, breaking down the application into smaller, independently scalable services using containerization (e.g., Docker) and orchestration (e.g., Kubernetes) for efficient management.

Scaling:

- Vertical Scaling: Consideration of hardware upgrades to increase server processing power and memory, suitable up to a certain point.
- Horizontal Scaling: Addition of more servers to the infrastructure for load distribution, sustainable for high traffic levels.

C) Problems that we could face while changing our design

Increased Complexity: Introducing scalability measures and distributed architecture can increase the overall complexity of the system. Managing multiple servers, communication between microservices, and data synchronization across distributed components can add complexity to the development, deployment, and maintenance processes. It requires a strong understanding of distributed systems and effective coordination between different teams.

Data Consistency and Integrity: In a distributed architecture, ensuring data consistency and integrity can become more challenging. As data is spread across multiple servers or microservices, maintaining consistency during updates and handling failures or network partitions becomes crucial. Implementing proper mechanisms like distributed transactions or event sourcing can help address these challenges.

Network Latency and Communication Overhead: In a distributed system, communication between different components may introduce network latency and communication overhead. As the user base grows, the increased network traffic can impact the overall system performance. Implementing efficient communication protocols, optimizing network configurations, and minimizing unnecessary data transfers can help mitigate these challenges.

REFERENCES

[1] https://docs.cleura.cloud/howto/getting-started/enable-openstack-cli/
[2] https://docs.openstack.org/python-openstackclient/latest/configuration/index.html
[3] https://httpd.apache.org/docs/2.4/programs/ab.html
[4] https://prometheus.io/docs