Auto-Scaling Cloud Services with OpenStack

Srinika Rachaprolu
Telecommunication Systems
Blekinge Institute of Technology
Karlskrona, Sweden
srra24@student.bth.se

***Abstract*—This paper presents an automated solution for deploying, managing, and scaling a cloud-based service in OpenStack. The system operates in three phases—Deployment, where networks, routers, and nodes (service, proxy, and bastion) are provisioned with tagged resources; Operations, which dynamically adjusts node count based on real-time monitoring; and Cleanup, ensuring efficient resource deallocation. The proxy node load-balances traffic (TCP/5000 for the Flask service, UDP/6000 for SNMP), while the bastion host enables SSH access and node health checks via ICMP.**

Performance is evaluated using Apache Benchmark, measuring mean response times and standard deviation across 1–5 nodes to assess scalability. Deployment time metrics and statistical significance are analyzed, followed by a discussion on large-scale challenges, including network bottlenecks and multi-location service distribution. The study identifies key optimizations for maintaining performance and reliability in distributed cloud environments.

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

I. Introduction

OpenStack is an open-source cloud computing platform that provides infrastructure as a service, enabling organizations to build and manage scalable cloud environments. Originally developed through a collaboration between NASA and Rackspace, it has grown into a comprehensive solution for deploying virtualized resources. The platform's flexibility makes it suitable for both private and public cloud implementations, offering components that handle computing, storage, and networking needs. [1]

At the heart of OpenStack's compute capabilities is Nova Compute, the service responsible for managing virtual machine instances. Nova allows users to provision compute resources dynamically, supporting both custom configurations and predefined hardware profiles known as flavors. These flavors define the allocation of processing power, memory, and storage for virtual machines. In this project, we utilize Ubuntu 20.04 as the operating system with a standard flavor configuration of one virtual CPU, 2GB of RAM, and 50GB of disk space to ensure consistent performance across deployed instances.

The process of creating and configuring virtual machines involves several coordinated steps. First, the system selects an appropriate hardware profile based on requirements. Then it retrieves and installs the specified operating system image. Additional storage volumes can be attached if needed, and network interfaces are configured to establish connectivity. This automated provisioning enables rapid deployment of resources while maintaining consistency across instances.

Our implementation leverages these OpenStack capabilities to create an automated solution with three distinct operational phases. The initial deployment phase establishes all necessary infrastructure components including networks, routers, and virtual machines. These machines include application nodes for running services, a proxy server to distribute incoming traffic, and a bastion host that provides secure access and monitoring capabilities. During normal operations, the system continuously monitors node availability and automatically adjusts the number of active instances to match demand. The final cleanup phase ensures efficient resource management by removing all allocated components when they are no longer needed.

To evaluate the effectiveness of this approach, we examine several key aspects. The design decisions behind the architecture are analyzed, considering alternative approaches and their trade-offs. Performance metrics are collected under varying conditions to understand how the system behaves with different numbers of active nodes. These tests measure both application responsiveness and deployment efficiency. Finally, we explore how the solution might scale in larger environments, identifying potential challenges and their impact on overall system performance. This comprehensive evaluation provides insights into both the current implementation's capabilities and directions for future improvements.

II. Initial Deployment Stage

The initial deployment stage automates the creation of cloud infrastructure in OpenStack, including networks, security groups, virtual machines, and configurations. All resources are tagged for identification and management.

1) Environment Setup and Authentication The script begins by loading OpenStack credentials from the provided openrc file to authenticate and establish a connection with the cloud environment. Using the OpenStack SDK or CLI commands, it interacts with the cloud API to provision and manage resources.

2) SSH Key Pair Configuration A new SSH key pair is generated in OpenStack (or an existing one is reused) to enable secure access to deployed instances. The private key

Identify applicable funding agency here. If none, delete this.

is saved in the local .ssh directory with a .pem extension and restricted permissions (chmod 600). This key facilitates SSH access to the Bastion, Proxy, and Service nodes during setup and operations.

3) Network and Security Infrastructure A private network and subnet are created to host internal nodes, while a router connects this network to the public cloud gateway for external access. Security groups are configured with rules allowing:

a)SSH (TCP/22) for administration,

b)HTTP (TCP/5000) for the Flask service,

c)SNMP (UDP/6000) for monitoring,

d)ICMP for node availability checks.

4) Instance Deployment Bastion Host: Deployed with a floating (public) IP, it acts as a jump host for SSH access to internal nodes and runs a monitoring agent to validate node availability via ICMP.

Proxy Node: Assigned a floating IP,it runs HAProxy/NGINX to load balance traffic across service nodes,listening on TCP/5000 (Flask)and UDP/6000 (SNMP).

Service Nodes: Three nodes are initially deployed in the private network (without public IPs), hosting the service.py Flask app and SNMP daemon. They are auto-registered with the Proxy for load balancing.

5) Floating IP Management To minimize costs, the script first checks for unassigned floating IPs before allocating new ones. The assigned public IPs (Bastion and Proxy) are stored in a configuration file for later reference.

6)Dynamic Configuration Generation [2] Configuration files are dynamically generated for:

HAProxy/NGINX (lists backend service nodes),

Monitoring Agent (Bastion checks node status),

SNMP (service monitoring). Internal IPs of service nodes are fetched via the OpenStack API and updated in the Proxy's load balancer config.

7)Service Installation and Setup The service.py Flask application and SNMP daemon are installed on all service nodes. The Proxy node is configured to forward requests to backend services, while the Bastion host schedules node availability checks every 30 seconds.

8)Resource Tagging and Verification All resources (instances, networks, security groups) are tagged with a unique identifier for tracking. A final verification ensures correct deployment before transitioning to the Operations Phase.

III. Ansible playbook

The playbook automates deployment and management of OpenStack cloud resources with three operational modes (deploy/operate/cleanup). It uses a unique TAG for all created resources and implements the required 3-service-node+PROXY+BASTION architecture. [3]

1)Base System Configuration All nodes receive essential package updates and security hardening. SSH access is configured for secure communication between nodes, with firewall rules restricting traffic to only necessary ports. Common monitoring agents are installed for baseline visibility.

2)PROXY Node Configuration HAProxy handles TCP/5000 load balancing for service.py using round-robin algorithm with active health checks. Nginx manages UDP/6000 traffic for SNMP via its stream module. Floating IPs are intelligently reused when available to minimize costs.

3)Service Nodes Configuration The Python service (service.py) runs as a managed systemd service with auto-recovery. SNMPd is configured with secure community strings and access controls. Both services expose health endpoints for monitoring.

4)BASTION Host Configuration Acts as secured SSH jump host with key-based access. Performs node monitoring via ICMP pings every 30s and service health checks. Maintains dynamic node inventory and triggers scaling actions when needed.

5)Monitoring and Auto-scaling Compares actual nodes against servers.conf every 30s. Uses OpenStack API to add/remove nodes as required. Collects performance metrics from all components to inform scaling decisions.

6)Cleanup Operations Systematically removes all tagged resources in proper order (instances→IPs→routers→networks). Verifies complete environment teardown and releases all allocated resources. [4]

IV. Operations Mode

The script continuously monitors and adjusts the environment to maintain the required number of service nodes (initially 3) from 'servers.conf'. It checks existing nodes via OpenStack, provisions new ones if needed, or removes excess ones. After changes, it updates configurations, waits 30 seconds, and deploys services ('service.py' + SNMPd) via Ansible.

The Bastion host verifies node availability via ICMP pings, while the Proxy node dynamically updates its load balancer. The script repeats checks every 30 seconds, ensuring stability while keeping resources optimized.

V. Clean up

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

VI. 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.

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

2) Scalability: Continuously monitoring the number of ac tive development servers, the operation stage script au tomatically adjusts their count in accordance with the servers.conf

file. This scalability feature enables the net work to accommodate a variety of demands without the need for manual intervention.

3) 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.

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

5) 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.

6) 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. [5]

VII. Alternatives considered

Several alternative approaches were considered before finalizing the current cloud deployment solution. Manual configuration was quickly ruled out due to its error-prone nature and inability to scale efficiently. While tools like Puppet, Chef, and SaltStack offered robust configuration management, their complexity and infrastructure requirements made Ansible a more suitable choice for this project. Ansible's agentless architecture and simplicity aligned better with our need for rapid, repeatable deployments.

Containerization with Docker and Kubernetes presented an appealing alternative, particularly for its efficiency and auto-scaling capabilities. However, the additional orchestration complexity and challenges with SNMP monitoring in containerized environments led us to maintain a traditional VM-based approach. Similarly, while Terraform's infrastructure-as-code model offered excellent consistency, we found direct OpenStack API calls through Python scripts provided greater flexibility for our custom monitoring and scaling needs.

We also evaluated OpenStack's native Heat orchestration service, which integrates seamlessly with the platform. Heat's template-based approach and auto-scaling features were compelling, but we needed more granular control over scaling decisions than Heat templates could provide. The learning curve for Heat's syntax further reinforced our decision to use custom scripts. Configuration templating with Jinja2 was considered but ultimately deemed unnecessary given our relatively static service configurations.

Each alternative presented unique advantages, but our chosen approach using Python scripts, Ansible, and direct OpenStack API calls offered the best balance of control, simplicity, and scalability. The solution provides the necessary automation while maintaining flexibility for custom monitoring logic and dynamic scaling decisions. This combination proved most effective for meeting the project's requirements of reliable deployment, operation, and cleanup in an OpenStack environment.

VIII. Conclusion

This project successfully designed and implemented an automated cloud solution on OpenStack, integrating deployment, operations, and cleanup phases into a cohesive system. The deployment phase efficiently provisions service nodes, a proxy, and a bastion host while optimizing public IP usage. During operations, the system dynamically scales nodes based on real-time monitoring, ensuring high availability through the bastion host's health checks. Finally, the cleanup phase ensures cost-effective resource management by terminating all allocated cloud assets. The solution demonstrates effective automation, leveraging load balancing and monitoring to maintain reliability while adhering to infrastructure-as-code principles.

IX. Scalability Challenges

While the solution performs well in small-scale deployments, scaling globally introduces network and service challenges. The proxy and bastion nodes become single points of failure, requiring redundancy for high availability. Multi-region deployments would face latency issues, necessitating distributed load balancing and synchronized configuration management. Additionally, centralized monitoring via the bastion host may struggle under heavier loads, suggesting a shift to decentralized monitoring tools like Prometheus for large-scale operations.

X. 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 di rectly influences user experience and service availability

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 pro tocols.

Latency: The distance between locations might cause latency, which may lead to delayed reaction times.

References

[1] T. Rosado and J. Bernardino, "An overview of openstack architecture," in *Proceedings of the 18th International Database Engineering & Applications Symposium*, IDEAS '14, (New York, NY, USA), p. 366–367, Association for Computing Machinery, 2014.

[2] L. Pramono, R. Cokro, and Y. Waskito, "Round-robin algorithm in haproxy and nginx load balancing performance evaluation: a review," pp. 367–372, 11 2018.

[3] F. Mbarek and V. Mosorov, "Load balancing algorithms in heterogeneous web cluster," in *2018 International Interdisciplinary PhD Workshop (IIPhDW)*, pp. 205–208, 2018.

[4] L. Chen, M. Xian, and J. Liu, "Monitoring system of openstack cloud platform based on prometheus," in *2020 International Conference on Computer Vision, Image and Deep Learning (CVIDL)*, pp. 206–209, July 2020.

[5] V. Marella, "Implementing infrastructure as code (iac) for scalable devops automation in hybrid cloud," *Journal of Sustainable Solutions*, vol. 1, pp. 145–153, 12 2024.