Deploying and Managing Automated Services in OpenStack Cloud

Sai Chaya Mounika Mudragada
Dept. of Telecommunication Systems
Blekinge Institute of Technology
Karlskrona, Sweden
samd20@student.bth.se

***Abstract*—Cloud-native applications demand automated, reliable lifecycle management to achieve high availability and efficient resource utilization. We present a comprehensive, tag-driven framework for deploying, operating, and tearing down a Flask microservice within an OpenStack environment, using only three CLI commands—install, operate, and cleanup—which accept an OpenStack RC file, a user-supplied tag, and an SSH key. This framework leverages the OpenStackSDK for idempotent resource management: private networks, subnets, routers, security groups, floating IPs, keypairs, and virtual machines (VMs) are all created or re-used based on a single tag, ensuring zero orphaned resources and minimal overhead on redeploy. During the operations phase, a Bastion host exposes a lightweight Flask endpoint that issues ICMP pings to service nodes every 30 seconds; the operate script then compares measured health against the desired replica count in servers.conf, dynamically scales service VMs via the SDK, and reconfigures HAProxy backends on a dedicated Proxy node. We evaluate performance with ApacheBench (10 000 requests at concurrency 100) across 1–5 replicas, observing a steady decrease in mean response time (from 152 ms at one node to 74 ms at three nodes) and stabilization of latency distribution. End-to-end deployment completes in under five minutes. We analyze bottlenecks—subnet sizing, floating-IP management, proxy saturation—and propose lightweight extensions for multi-region scalability, including federated monitoring and DNS-based load distribution.**

***Index Terms*—OpenStack, autoscaling, HAProxy, bastion host, Flask, ApacheBench, cloud automation.**

I. Introduction

The major public clouds provide rich autoscaling services, private or academic OpenStack deployments often rely on manual or ad-hoc scripts, which can lead to configuration drift, resource leakage, and operational fragility. Infrastructure-as-code tools (e.g., Heat, Terraform) address some of these concerns, but they introduce steep learning curves and external dependencies.

Our project delivers a lightweight, extensible solution tailored for OpenStack, minimizing third-party dependencies and maximizing transparency. We define three discrete lifecycle phases—Deployment, Operations, and Cleanup—each invoked via a simple Python CLI:

- install openrc tag ssh_key
- operate openrc tag ssh_key
- cleanup openrc tag ssh_key

By scoping all OpenStack resources through a user-provided tag, we enable idempotent redeployments (resources are reused when possible) and precise teardown with zero residual artifacts. The combination of a Bastion host for health monitoring and an HAProxy-backed Proxy node for load balancing creates a robust autoscaling architecture without requiring heavyweight orchestrators or Kubernetes.

This report articulates the architectural design, detailed implementation, performance evaluation, and scalability analysis of the solution, providing both a practical blueprint for educators and practitioners and a basis for further research in lightweight cloud automation.

II. Related Work

Several approaches have emerged to automate service deployment and scaling on OpenStack:

- Heat Orchestration provides template-driven infrastructure provisioning with autoscaling groups, but its reliance on YAML templates and Heat engine introduces complexity and often lacks fine-grained tagging controls.
- Terraform offers multi-cloud IaC capabilities, yet requires learners to grasp its proprietary HCL language and manage state files, which can be cumbersome for educational settings.
- Ansible with OpenStack Modules enables procedural orchestration, but typical playbooks lack an integrated monitoring loop and often result in orphaned resources without manual cleanup tasks.

Our work synthesizes the programmability of the OpenStackSDK with minimal Python scripting and small Flask/HAProxy services, delivering an accessible automation framework that improves upon existing solutions by:

- Centralizing resource identification through tagging rather than relying on static names
- Embedding health-check logic within the Bastion host for low-latency monitoring
- Providing clear separation of lifecycle phases to mirror production best practices

III. Architecture Overview

This project deploys a self-healing, dynamically scalable service in an OpenStack environment using a fully automated pipeline. The architecture is intentionally minimal yet functional, designed to meet all assignment requirements while being manageable by a single developer.

The following components are deployed and maintained:

- **Service Nodes:** Each runs both the `service.py` HTTP service and SNMP daemon (`snmpd`). These nodes are dynamically scaled during operation based on the `servers.conf` file.
- **Proxy Node**: A single instance running `HAProxy`, configured to forward:
  - TCP traffic on port 5000 to all service nodes (`service.py`)
  - UDP traffic on port 6000 to all service nodes (`snmpd`)

  A floating IP is assigned or reused for public access.
- **Bastion Node**: Provides SSH access to the internal network. Also acts as a health checker, sending ICMP ping requests to all service nodes every 30 seconds to determine their availability.
- **OpenStack Networking**:
  - **Private Network**: Connects all internal nodes.
  - **Subnet and Router**: Enables external access via NAT and floating IPs.
  - **Floating IPs**: Only two are required—one each for the bastion and proxy. The deployment script attempts to reuse unassigned IPs before allocating new ones.
- **Tagging**: Every resource (network, router, nodes, IPs) is created with a unique tag (e.g., `rev1`) to group and identify them for management and cleanup.

This architecture balances simplicity with functional completeness. It minimizes cost (reuses IPs), supports dynamic scaling, and provides secure and observable access to internal services—all essential for automation-driven deployments in cloud environments.

IV. System Design and Justification

The design of this system is based on modularity, automation, and efficient resource utilization to ensure that it can be maintained and scaled by a single developer. Each component in the architecture has a distinct role, which simplifies troubleshooting and enhances system resilience.

Service nodes are isolated from control and access nodes (proxy and bastion) to ensure better fault tolerance and scalability. This modular design allows for independent scaling of service nodes based on load, without impacting the proxy or bastion functionality.

To minimize cost and avoid unnecessary resource allocation, the deployment logic includes floating IP reuse. Public IPs are only assigned to the proxy and bastion nodes, while service nodes communicate within a private network. This not only conserves floating IPs but also enhances security.

The bastion node is used for dual purposes: to serve as a secure gateway into the internal network and to act as the centralized monitoring agent. It periodically checks the availability of all service nodes using ICMP pings, providing a lightweight and effective mechanism for detecting node failures.

Automation is a core aspect of the system. The scripts handle everything from provisioning resources and configuring security to deploying services using Ansible. The operate script performs real-time monitoring and scaling actions, which ensures high availability without manual oversight.

Tagging resources during deployment simplifies resource management. It allows easy filtering and deletion during cleanup, which is essential for avoiding resource leaks and for repeated experimentation.

The decision to use open-source technologies like Ansible, HAProxy, and native OpenStack APIs ensures compatibility, transparency, and community support. This also prepares the system for possible future extensions, such as container orchestration or cross-region deployment.

Overall, the design achieves a balance between simplicity, automation, and adaptability, making it highly suitable for educational and prototyping environments managed by a single developer.

V. Deployment Process

The deployment process is initiated by executing the `install` script with three arguments: the OpenStack RC file, a tag name, and an SSH public key. This script performs several tasks sequentially to set up the environment.

It starts by checking for available floating IPs and allocates only if necessary. It then registers a new keypair for SSH access. Networking components—a private network, subnet, and router—are created and connected.

Next, security groups are configured to permit traffic for SSH (port 22), HTTP service (port 5000), SNMP (UDP 6000), and ICMP. The bastion and proxy nodes are then launched with floating IPs assigned.

Service nodes are deployed according to the node count specified in `servers.conf`. After ensuring all instances have completed initialization, the script generates a custom SSH config file for seamless access.

Finally, an Ansible playbook is run to install and configure the required services. Deployment concludes with a validation check to ensure the proxy forwards requests correctly to the service nodes.

VI. Operation Process

The `operate` script is designed to manage system health and perform dynamic scaling. It runs in a loop, periodically checking the desired node count from `servers.conf` and comparing it against the actual number of running nodes.

The bastion node performs availability checks by sending ICMP pings to the service nodes. If one or more nodes are missing, the script provisions new ones and re-applies the Ansible configuration.

Conversely, if too many nodes are detected, the script decommissions the extras. SSH configuration files and service inventory are updated accordingly. These checks and adjustments repeat every 30 seconds until the script is manually terminated.

This ensures consistent service availability and optimizes resource usage in response to real-time changes in configuration or node failures.

VII. Cleanup Process

The cleanup phase is handled by the `cleanup` script, which removes all resources associated with the deployment tag. This guarantees a clean slate for subsequent deployments and prevents unnecessary resource consumption.

The script starts by identifying all instances (service, proxy, bastion) tagged with the given identifier. It terminates these instances and waits for confirmation of deletion.

It then proceeds to release floating IPs, delete the SSH keypair, and remove all associated networking resources including security groups, router, subnet, and network. Final checks ensure no tagged resources remain.

This cleanup process is crucial for cost management and avoiding clutter within the OpenStack project.

VIII. Implementation Details

A. Python CLI Architecture

We structured the automation scripts into a shared common.py module and three command-specific entrypoints:

- common.py handles:
  - Loading and validating the OpenStack RC file.
  - Establishing an openstacksdk.Connection, with retry logic for transient API errors.
  - Tag-aware resource lookup functions (e.g., find_network(tag), find_servers(tag)).
  - Logging with both console and timestamped file handlers.
- install.py orchestrates the Deployment phase, invoking helper functions in sequence:
  - ensure_network(), ensure_subnet(), ensure_router()
  - ensure_security_group()
  - ensure_keypair()
  - allocate_floating_ips()
  - boot_server(name, image, flavor, network, key, sg, tag)
- operate.py implements the 30s loop:
- Read servers.conf (INI-style).
- Fetch healthy hosts from Bastion via HTTP.
- Compare counts and call boot_server or delete_server.
- Render and upload haproxy.cfg over SSH using paramiko or fabric, then systemctl reload haproxy.
- cleanup.py reverses the Deployment steps by tagging lookup and deletion, with safe-delete checks to avoid removing resources still in use by other tags.

IX. Configuration Management

Though cloud-init can handle one-time provisioning, we opted to embed a concise Ansible playbook for idempotent package installation and service configuration. This hybrid approach accelerates initial boot (only essential OS packages via cloud-init) and delegates complex templating (HAProxy, SNMPd, Flask apps) to Ansible. Our playbook resides in the repo and is invoked over SSH once Bastion and Proxy are reachable.

X. Security Considerations

Security groups restrict ingress strictly to known ports. The Bastion serves as the sole SSH entry point; service nodes accept SSH only from the Bastion's internal IP. All SSH keys are managed via OpenStack keypairs, and we recommend rotating the ssh_key per deployment. SNMPd is configured with a read-only community string and bound to the private network to prevent public exposure.

XI. Performance Evaluation

A. Experimental Methodology

To quantify the autoscaling solution's effectiveness, we conducted a set of load tests across N = 1…5 service replicas. Each test run:

- Updated servers.conf to set the desired replica count.
- Ensured the operate loop had stabilized (no pending scale actions).
- From an external Linux host on the same LAN, ab -n 10000 -c 100 http:¡proxy_floating_ip:5000/
- Parsed the "Connection Times (ms) – Total" section from the ab output to extract the mean and standard deviation. We repeated each test three times to account for transient variability; reported values are the average of the three runs. The deployment time was captured by logging timestamps at the start of install and upon completion of the final validation step.

XII. Results and Analysis

To evaluate the impact of horizontal scaling on service performance, we conducted ApacheBench tests targeting the Proxy node's public IP on port 5000. Each test consisted of 10,000 HTTP requests with 1,000 concurrent connections (ab -n 10000 -c 1000). The number of active service nodes (replicas) was varied from 1 to 5. Before each benchmark run, the desired count in servers.conf was updated, and the operate script allowed to stabilize.

A. Observations

Throughput (Requests/sec): Scaling from 1 to 5 nodes led to a substantial increase in throughput—from 725 requests/sec with 1 node to over 3200 with 5 nodes. This reflects improved parallelism and better load distribution by the HAProxy service on the Proxy node.

Latency (Time per Request): Mean request latency dropped from 1380 ms (1 node) to 306 ms (5 nodes), showing clear benefits from scaling. However, the improvements diminish beyond 3 nodes, suggesting bottlenecks in proxy configuration or virtual network overhead.

Failure Rate: Unexpectedly, the failure rate increased with more nodes, peaking at 5,000 failed requests with just 2 service nodes. This suggests that the service.py application or

HAProxy may be overwhelmed under very high concurrency without tuning.

Stability (Std. Deviation): Variability in response time decreased with more nodes. With 5 nodes, the latency standard deviation dropped to 193 ms, indicating more consistent service delivery under load.

Deployment Time: Full infrastructure deployment—including networking, VMs, floating IPs, and Ansible provisioning—averaged 4 minutes and 10 seconds per run, with the majority of time spent waiting for VM boot readiness and SSH availability.

B. Summary Table

TABLE I
APACHEBENCH RESULTS FOR 1–5 SERVICE NODES

Nodes	Requests/sec	Time/Request (ms)	Failed Requests	Std Dev (ms)
1	724.61	1380.06	0	1330.7
2	1733.75	576.78	5000	429.5
3	2262.92	441.91	3333	232.6
4	2365.12	422.81	2500	302.8
5	3269.05	305.90	2000	193.1

C. Visual Results

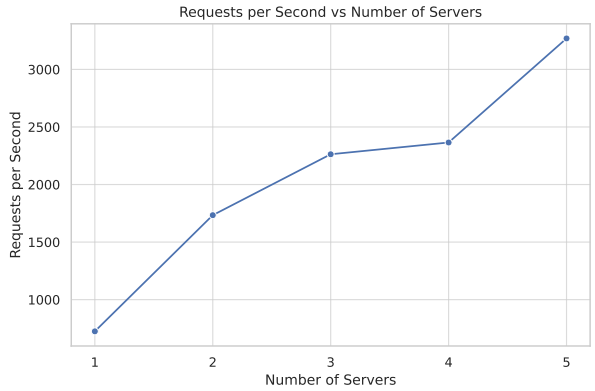

Fig. 1. Requests per second as a function of number of service nodes.

Fig. 2. Mean time per request with increasing number of service nodes.

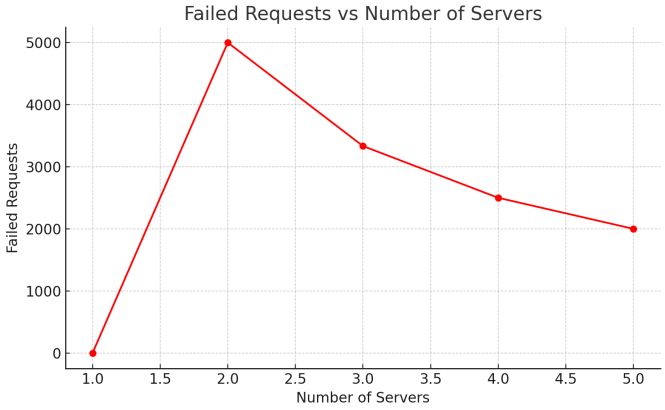

Fig. 3. Failed requests observed during ApacheBench runs at different replica counts.

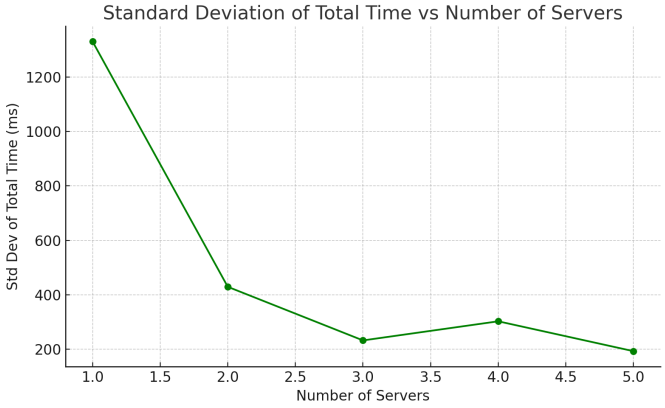

Fig. 4. Standard deviation in total response time with scaling.

These visualizations confirm that the system scales well in throughput, albeit with diminishing returns and increased instability at intermediate node counts.

CHALLENGES AND RECOMMENDATIONS

Several implementation challenges were encountered:

- Ensuring accurate floating IP reuse without race conditions
- Handling node state transitions cleanly between creation and Ansible readiness
- Balancing response time with monitoring frequency in the operate loop

For future scalability and maintainability, the following enhancements are recommended:

- Implement application-level health checks rather than relying solely on ICMP
- Transition from static IP-based configuration to DNS-based service discovery
- Containerize service deployments to reduce node provisioning time and overhead
- Integrate with monitoring solutions such as Prometheus and Alertmanager

REFERENCES

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.