A Comparative Analysis of OpenStack Autoscaling Engines: Evaluating Performance and Scalability, and Usability of Heat and Senlin under Real-World Workload Patterns

Liyanage Dinith Ishanka Appuhami

239305R

Master of Science in Computer Science

Department of Computer Science and Engineering

Faculty of Engineering

University of Moratuwa

Sri Lanka

January 2025

A Comparative Analysis of OpenStack Autoscaling Engines: Evaluating Performance, Scalability, and Usability of Heat and Senlin under Real-World Workload Patterns

Liyanage Dinith Ishanka Appuhami

239305R

Thesis submitted in partial fulfillment of the requirements for the degree.

Master of Science in Computer Science

Department of Computer Science and Engineering

Faculty of Engineering

University of Moratuwa

Sri Lanka

January 2025

DECLARATION

I declare that this is my own work, and this thesis/dissertation does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any other University or Institute of higher learning, and to the best of my knowledge and belief, it does not contain any material previously published or written by another person except where the acknowledgment is made in the text. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: Date: 21.04.2025

The above candidate has carried out research for the master's thesis/dissertation under my supervision. I confirm that the declaration made above by the student is true and correct.

Name of Supervisor: Dr. M.P.A.P. Wijayasiri

Signature of the Supervisor: Date: 21.04.2025

DEDICATION

I dedicate this thesis to the individuals who have helped me along the way in my academic career. I appreciate seeing this journey through to the very finish.

ACKNOWLEDGEMENT

First, thanks go to my supervisor, Dr. Adeesha Wijayasiri, for his guidance during this thesis project. He always provided thorough feedback when I needed it. I consider myself very lucky to have had the chance to work under his guidance.

Also, I'm grateful to all the MSc in CS academic staff who taught me different subjects in computer science.

Furthermore, I would like to thank the network and data center teams of Srilankan Airlines IT for allowing me to use their infrastructure for practicing and testing my work.

Finally, I would like to thank my family for their encouragement and understanding throughout this work.

ABSTRACT

Cloud computing is a demanded field that opens the door for various possibilities due to its characteristics such as flexibility and on-demand availability. Basically, it presents an infrastructure for different kinds of services and tools via the public internet [1]. Auto-scaling, that is allocating and deleting resources without the involvement of the user, is one of the principal features in the domain of cloud computing. [2]. In domain of autoscaling on cloud systems, the demand for autoscaling systems, applications and services continues to surge, there is an increasing need for flexible and cost-efficient solutions to handle dynamic traffic patterns. Most of the time private cloud based autoscaling solutions are not used, because of the absence of proper information of the private cloud based autoscaling solutions and their performance. Specially in OpenStack who is the leading private cloud provider in the domain of cloud computing. As a result of this lack of knowledge and performance information of the private cloud based autoscaling solutions, the community are still using manual scaling methods when necessary or they are using public cloud based autoscaling solutions. However, there are several identified issues such as inefficiency and management challenges in manual scaling methods. When it comes to the public cloud based autoscaling solutions, there are set of challenges and cons for example, lack of cost-effectiveness, lack of transparency and administrative control, and vendor lock-in issues are few of them. As a solution for the above-mentioned issues, in this project we are giving the community an opportunity to get an idea about the OpenStack based autoscaling solutions (namely Heat and Senlin) and their behavior and performance against various kinds of practical workloads. As per our knowledge, there is not any previous literature that compares the different autoscaling solutions in OpenStack cloud. This work involves doing comprehensive research on the existing cloud-based auto-scaling solutions and the Zed version of fully functional OpenStack environment is implemented including the Heat and the Senlin projects. These autoscaling engines are tested with 10 different kinds of practical workloads which are generated using Apache JMeter and their performance metrics are recorded. These metrics are analyzed using MCDA method and the optimum autoscaling solution for each workload pattern is determined. We hope this work will contribute to the community who are interested in OpenStack based autoscaling to decide the proper autoscaling solution based on the nature of their workloads.

Keywords: Cloud computing, Auto-scaling, OpenStack, Heat, Senlin, MCDA

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	Description
VM	Virtual Machine
OS	Operating System
AWS	Amazon Web Services
GCE	Google Compute Engine
GCP	Google Compute Provider
VPS	Virtual Private Server
CMS	Content Management System
URL	Uniform Resource Locator
CPU	Central Processing Unit
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory network
AR	Auto Regressive
ARMA	Auto Regressive Moving Average
ES	Exponential Smoothing
API	Application Programming Interface
MQ	Message Queue
SLA	Service Level Agreement
MIMO	Multi-Input, Multi-Output
ML	Machine Learning
RT	Response Time
QPS	Queries Per Second
RDBMS	Relational Database Management System
MCDA	Multi-Criteria Decision Analysis

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

This chapter provides basic background information about cloud computing, virtualization, auto-scaling, OpenStack, and its components. In 1.1, a brief introduction is given about cloud computing. In 1.2, Cloud computing service models are discussed. 1.3 gives an overview of deployment models. 1.4 gives a brief introduction to virtualization in cloud computing. Then, Section 1.5 introduces auto-scaling in cloud computing and different auto-scaling techniques. Section 1.6 discusses an overview of the OpenStack platform and its main components. Finally, Section 1.7 discusses the need for auto-scaling in private clouds.

1.1 What is Cloud Computing?

Cloud computing is a delivery method for Internet-based services that provide access to the functions of computers, such as infrastructure, networking, and software. These offerings result in quicker results, adaptability in products, and economies of scale. Instead of a data center, the customers rent everything from applications to the infrastructure from their cloud service provider. Therefore, it allows consumers to do away with the costly upfront payment related to owning and maintaining their IT infrastructure, so they use cloud computing services and pay for what they use exactly as they use it. Subsequently, cloud computing service companies can also enjoy these huge economies of scale by providing the same services to different clients.[3].

1.2 Cloud computing service models

By removing the burden of day-to-day tasks, such as provisioning hardware, maintaining resources, and planning capacity, and in so doing, leaving IT teams and software developers to focus on more important things, cloud computing prospers. With this increase in adoption, service models and deployment options have arisen to satisfy a variety of user needs. Different cloud services and deployment types offer differing degrees of control, customization, and administration. An understanding of Infrastructure as a Service, Platform as a Service, and Software as a Service, along with their deployment models, gives the users ability to settle on an arrangement that's best for their particular aim.[4].

1.2.1 Infrastructure as a Service (IaaS)

Infrastructure as a Service (IaaS) enables cloud providers to offer fundamental computing resources over the Internet so that users can store data and run their applications as needed.[5]. The provisioning of these virtualized services like computing power, storage, and networking is carried out dynamically based on user demand criteria. That is, the provider controls the fundamental components

underneath, such as networking and the provisioning of operating systems, while customers exercise control over the software layers, such as middleware, application data, and runtime environments. Users are typically given virtual machines (VMs) with their chosen operating systems, and the provider places minimal restrictions on their use.[6]. At its core, the server is an IaaS solution for hosting virtual machines and deploying applications. Other services provided include flexible storage options and the ability to manage server functions such as VM creation, access permissions, and instance control.[7]. Some good examples of IaaS platforms are AWS, Microsoft Azure, Google Compute Engine (GCE), Linode, Rackspace, and Cisco Metapod.

1.2.2 Platform as a Service (PaaS)

PaaS provides developers with a complete environment for creating, testing, and deploying applications directly into the provider's cloud ecosystem. Such users are generally developers who use the platform to design and deliver applications to end users. Towards its end, PaaS is a higher abstraction above IaaS because the cloud provider manages the essential components like the middleware and runtime environment. Though customers do not manage the core infrastructure of the cloud, they own and control their deployed applications and can also configure most features of the application environment according to their needs.[7]. Some of the notable PaaS examples are AWS Elastic Beanstalk, Heroku, Google App Engine, Windows Azure, Apache Stratos, Red Hat OpenShift, and services offered in Salesforce's platform.

1.2.3 Software as a Service (SaaS)

With Software as a Service (SaaS), applications are delivered over the internet, thus eliminating the need for users to install software on their local machine or on their internal servers. This is the most commonly adopted cloud model and the one most often encountered in everyday life. In this model, the cloud provider manages everything from networking and server infrastructure to the applications themselves. Users do not have control or visibility into the underlying systems but can typically modify user-level settings within the application. SaaS applications are usually accessed using lightweight clients like web browsers, which makes them independent of the device and available from just about anywhere. Due to the ongoing paradigm shift with users slowly distancing themselves from traditional software installations, the SaaS acts as a cost-efficient and tangible alternative. It facilitates rapid service deployment by application providers with very little capital expenditure on hardware. Real-time collaboration is supported by many of the SaaS platforms. This means that now multiple users can work on a document simultaneously and share it with one another [7]. Popular SaaS offerings include Gmail, Office 365, Google Docs, and Cisco Webex.

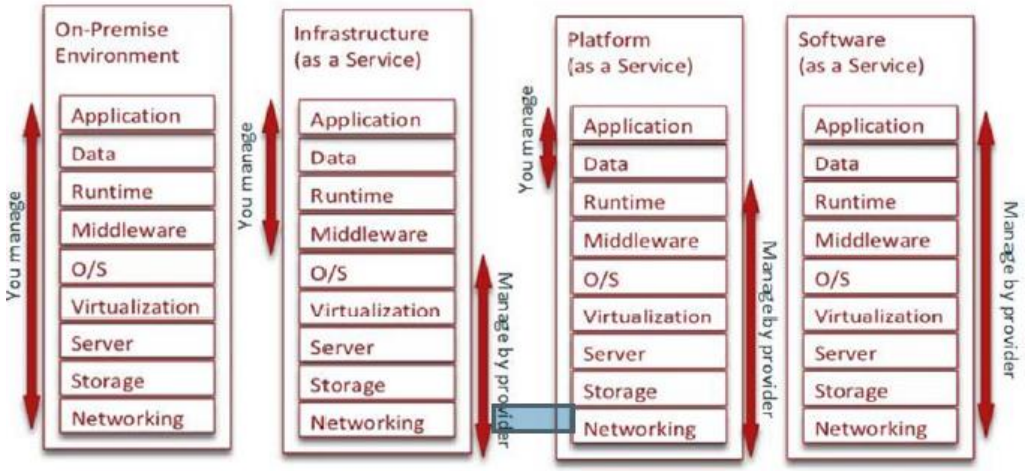

Fig. 1.1: Server stack comparison between on-premises infrastructure, IaaS, PaaS, and SaaS [9]

1.3 Cloud computing deployment models

A cloud deployment model defines all the factors that constitute a cloud environment, including ownership, access level, scale, and intended use. It will tell how the cloud servers are placed and controlled by whom. A deployment model will also denote how the infrastructure is organized, which aspects are customizable, and whether the user would receive pre-built services or configure things independently. These models provide a precise definition of the nature of interaction between cloud infrastructure and its users. Now, here are the various types of cloud deployment models as discussed below [8].

1.3.1 Public Cloud

Public clouds refer to deployment models where computing resources and services are made available to the general public via the internet. Because of the wide range of users they support, public clouds may pose security challenges that private setups would not account for. Here, the infrastructure is in possession and is managed by the cloud service provider rather than an end user. Public cloud platforms are mostly adopted for their ease of access and flexible service delivery. They offer scalable computing resources to multiple customers, free or via subscription, or charged on-demand basis. One of the more famous public cloud services is Google App Engine. [8].

1.3.2 Private Cloud

It is a deployment model that is meant to be built for a single organization or user, as opposed to the public cloud. Here, resources are not available to others, thus offering a controlled and dedicated computing environment. Commonly referred to as the internal cloud, it allows access to systems and services available within the boundary of a specific organization. The infrastructure is usually hosted in a secure environment

and is managed by the organization's own IT department, with all possible strong defenses established through firewalls. This model provides enhanced flexibility and greater control over the resources of the cloud.[8]. In this study, we specifically focus on the private cloud model implemented using OpenStack.

1.3.3 Hybrid Cloud

As such, hybrid cloud computing connects private clouds and public clouds through a proprietary layer to ensure that organizations benefit from the best of both worlds. This configuration allows applications to run in secure and regulated environments while leveraging public cloud services for lower costs and high scalability. Organizations have the capability of moving data across different cloud infrastructures and workloads, depending on their specific needs, by mixing and matching two or more deployment models.[8].

1.3.4 Community Cloud

The Community Cloud model affords multiple organizations with parallel purposes or operational needs access to shared systems and services. It is a distributed infrastructure that pools resources across different cloud environments in order to meet the common requirements of a specific group, industry, or sector. The underlying infrastructure is shared by interfacing organizations that share responsibilities or concerns. The cloud can be managed by a third party or jointly by the organizations involved.[8].

1.3.5 Multi-Cloud

While somewhat similar to the hybrid cloud approach of combining private and public clouds, multi-cloud focuses exclusively on various public cloud platforms without necessarily combining them with private infrastructure. They offer services to make relying on their service more effective, but outages still occur; it is just rare for multiple independent providers to fail at the same time. Therefore, it is reasonable to conclude that the adoption of a multi-cloud strategy would greatly enhance service availability and reduce the risks of downtime.[8].

1.4 Virtualization in Cloud Computing

Virtualization can be defined as the process of creating a virtual instance of some physical resource, such as a server, a desktop, a storage device, an operating system, or a network component. In other words, it is a technology that enables one or more users or organizations to share a single physical resource or application, assigning them logical names and allowing them access as required. In hardware virtualization, a virtual machine (VM) is set up on an existing operating system and hardware over which the VM can exercise more or less complete control. This sets up isolated environments with separate I/O between one another. The physical machine hosting

the virtual machine is referred to as the Host Machine, while the virtual machine itself is referred to as the Guest Machine. Virtualization really matters in cloud computing, as it allows users to share applications and data alongside the underlying infrastructure. One substantial benefit of virtualization technology is the increased efficiency in the management of software. So, when a new version of an application is released, the cloud providers can offer it via virtualization to their users while minimizing the cost of deploying such updates. Using third-party providers to manage servers and applications allows cloud services to scale at an economical price, with the third-party provider paying for the services on a monthly or yearly basis. [10].

1.5 Auto-scaling in cloud computing

Auto-scaling is a critical aspect of cloud computing, enabling dynamic adjustment of computing resources based on the workload demand. It provides the ability to scale resources up or down automatically, depending on factors such as traffic volume, application performance, and resource utilization. By implementing auto-scaling techniques, organizations can effectively optimize resource allocation, ensuring that the right amount of computing power is allocated at any given time, thereby avoiding over-provisioning or underutilization. One of the key advantages of auto-scaling is its potential to reduce costs. Traditional static provisioning of resources often leads to overprovisioning, resulting in wasted resources and increased expenses. Auto-scaling allows organizations to dynamically scale resources in response to demand, ensuring that resources are allocated as needed. This leads to cost savings by eliminating unnecessary resource allocation during periods of low demand and scaling up resources during peak usage.

Furthermore, auto-scaling plays a crucial role in enhancing availability and performance. By automatically adjusting resources, auto-scaling helps maintain consistent performance levels even during sudden spikes in traffic. It enables organizations to seamlessly handle increased user loads without service disruptions or degradation. With auto-scaling, applications can scale horizontally by adding additional instances to distribute the workload effectively, ensuring high availability and optimal user experience [11]. Let’s review more details and different auto-scaling techniques.

1.5.1 Classification of auto-scaling techniques

Two primary means of scaling systems are vertical and horizontal scaling. Vertical scaling typically refers to increasing/decreasing computing resources by upgrading to a larger or smaller virtual machine (VM) to fulfill performance needs. In contrast, horizontal scaling adds or removes multiple instances of VMs without a service interruption. The system continues operating as new instances are provisioned in response to fluctuating demand, and when that demand decreases, the extra resources are released. Most modern cloud environments support horizontal scaling. The categorization of the many proposed autoscaling strategies that appear in the academic

and industrial literature may not be an easy task, although the categorization in broad terms forms a general guide. However, a very basic method of classification involves dividing them into two types, usage-proactive and reactive: the reactive ones react to the real-time spike of demand with resource allocation, while the proactive types allocate resources beforehand using predictions for future conditions.[14].

This auto-scaling mechanism can use many approaches: predefined thresholds, reinforcement learning, fuzzy logic systems, queuing models, time-series forecasting, and other machine learning methodologies.[15].

1.5.1.1 Threshold-based Rules

Threshold methods apply to commercial auto-scaling systems, as they are easy to use and easy to understand in design. Generally speaking, they react to demanding incoming metrics measured, such as CPU, memory, or network usage, to combat unexpected changes in demand. This approach, however, may fall short in effectiveness due to instances being a fixed number. Beloglazov and Buyya [16] proposed an approach to this issue by dynamically changing threshold values for the consolidation of virtual machines (VMs), based on the adaptive CPU utilization levels of each VM running on a physical server.

1.5.1.2 Reinforcement learning

Autoscaling with reinforcement learning provides a forecasting mechanism, adapting to the decisions made regarding the system's behavior. This is the middle mechanism by which an auto-scaling agent engages in continuous interaction with the scalable application: the learning effect of the agent's experience over time develops knowledge of what scaling actions will be most proficient at that time. Because reinforcement learning doesn't depend on historical data, it is memoryless; thus, future states will only be determined from the current state alone. One of its particular types is Q-learning, which develops Q-function mapping states and actions through repeated interactions to finally lead the system toward a policy optimal for scaling.[17].

1.5.1.3 Fuzzy learning

Fuzzy control-based learning processes permit the generation of intuitive rules without formal definitions, allowing users to use linguistic terms to formulate knowledge. In creating the fuzzy controller, users usually have to define some IF-THEN rules for an auto-scaling behavior [18]. These manually defined rules can sometimes lead to inefficient or suboptimal scaling, aggravating costs on the part of cloud service providers. In an attempt to minimize manual rules, fuzzy-rule-based logic has been integrated into reinforcement learning by researchers. [19]. FQL4K is one of the approaches aimed at improving dynamic resource allocation. This fusion of fuzzy control mechanisms with fuzzy Q-learning techniques will help in achieving the goal. [20].

1.5.1.4 Queuing theory

One of the classic methods used for capacity planning is Queuing Theory [21], which indicates when a service system needs augmentation in capacity by assessing the mixed volume of queued service requests. The model assumes static system behavior with constant request arrival rates and processing rates. However, this sort of assumption truncates the applicability of this method in modeling the dynamic nature of cloud-based architecture. As queuing theory by itself can only give estimates for performance indicators, it is usually combined with other methods, such as threshold-triggered policies, control theory, and reinforcement learning, to solve auto-scaling problems more efficiently.

1.5.1.5 Time-series analysis

Time series data consists of measurements taken over regular time intervals, the variations in some of the metrics over time, thereby facilitating a temporal analysis of the data. In the cloud, good auto-scaling requires an appropriate time series analysis to reveal the trend in resource usage and process any fluctuation in demand as quickly as possible while being cost-effective. Common statistical forecasting methods for time series include Moving Average techniques, Auto-Regressive (AR) Methods, Auto-Regressive Moving Average (ARMA), Exponential Smoothing (ES), and the Auto-Regressive Integrated Moving Average model (ARIMA).[21].

1.5.1.6 Machine learning techniques

Machine learning methods can predict cloud workloads [22-24] such as Support Vector Machine, Neural Network, and Linear Regression. But one drawback with these is that events leave long-term historical data and do not compute moment time lags. It was Jeff Elman who invented the Recurrent Neural Network, which adds a memory capacity to conventional feed-forward networks, so that they can learn sequences with short time gaps (less than 10 steps). Unfortunately, RNNs cannot recall earlier events when there are longer time lags. Thus, Long Short-Term Memory (LSTM) networks, which are a specialized type of recurrent neural network, were developed to work with longer sequences and are able to learn time lags of up to 1000 time steps between associated events.

1.5.2 Available cloud-based autoscaling in the industry

The major auto-scaling features of the three clouds, Google Cloud Platform (GCP), Amazon Web Services (AWS), and Microsoft Azure-are described, based on horizontal scaling of virtual machines; most vertical scaling techniques in the industry are still in early stages of development. The efficacy of an auto-scaling system is determined by the speed of action, namely the time taken to begin or conclude a scaling process. In the cloud platforms considered, it typically requires some one full minute for the initiation and termination of virtual machines.[98].

1.5.2.1 Rule-based autoscaling

Both AWS and Azure offer auto-scaling features based on predefined rules, referred to as simple scaling in AWS [99] and AutoScale in Azure [100]. Before automating scaling, it is recommended to set up a threshold for a certain metric, i.e., CPU utilization < 80%. Since the metric is below the threshold, a scaling operation kicks off (For example, launching a VM). AWS does this through CloudWatch, its monitoring service. With CloudWatch, users can monitor the various metrics, either system-wide (for example, CPU utilization) or user-defined ones. Continuous collection of metric data, with data point creation based on aggregation over fixed intervals (usually 1-2 minutes). Additionally, CloudWatch allows users to create alarms that can trigger based on a defined threshold on metrics. Similar functions are also provided by Azure Monitoring, where users can monitor a number of metrics and configure alerts based on them. [98].

Whenever an alert goes off, it executes the predetermined policy. This policy can include the addition or removal of some number of virtual machines, say, adding 3 VMs or scaling according to the current size of the system, as in the case of adding them by 20%. After scaling, this system typically enters a cooldown period of 180 seconds (by default) that allows the system to stabilize and prevents multiple scaling actions from occurring at the same time.[98].

1.5.2.2 Step scaling

The step scaling policy presents an improvement over the AWS rules based scaling system concept [99]. Where users can define a policy table that identifies the scaling actions to take when certain states of affairs exist. For example, for CloudWatch alerts, it could be specified what numbers or percentages of virtual machines (VMs) would be added or removed once some metric threshold has been crossed. For instance, Table 1.1 establishes how a set of step scaling rules can be based on CPU utilization, the metric UCPU, rather than having to state specific numbers of instances during scaling actions. What this holds for the alternate uses of such types of scaling is the warm-up time instead of a cool-down period, which is the time of waiting before the newly launched or shut-down VM can be considered in the policy action. So, it allows multiple actions through policy, but it only counts those actions once the warm-up time has been exhausted.[98].

TABLE 1.1: EXAMPLES OF STEP SCALING RULES [98]

Values	Action
$U_{CPU} > 90\%$	$+20\%\ VMs$
$80\% < U_{CPU} \leq 90\%$	$+15\%\ VMs$
$70\% < U_{CPU} \leq 80\%$	$+10\%\ VMs$
$20\% < U_{CPU} \leq 30\%$	$-20\%\ VMs$
$30\% < U_{CPU} \leq 40\%$	$-15\%\ VMs$
$40\% < U_{CPU} \leq 50\%$	$-10\%\ VMs$

1.5.2.3 Target tracking scaling

AWS [101] and GCP [102] have developed a very advanced scaling methodology that is completely unbiased because the users are not required to input any complex rules or policies. Users will only have to specify a requested target value for a desired metric; automatically, this is within the scope of the auto-directed traffic. It is designed for that--to keep adjusting the number of virtual machine (VM) instances with an idea of allying a metric value as closely as possible to a target-and not much more. Both AWS and GCP do not reveal the algorithm used in formulating the approach, but both include cool-down and warm-up periods to ensure smoothness and to guard against some paradoxical moves.[98].

1.5.2.4 Scheduled scaling

Scheduled scaling is offered by AWS [103], Azure [104], and GCP [105] to allow users to set time-based triggers for scaling actions. For example, it could be creating more VMs over weekends and reducing them on Mondays. Predictive scaling is an enhanced version of this offered by AWS [106]. It studies workload patterns of the preceding 14 days and anticipates what VMs would be required in the next 2 days. Based on this analysis, it automatically schedules scaling actions. The algorithm or method used for this is not provided. Both scheduled and predictive scaling can be considered proactive actions; they usually serve as complements to reactive methods like rule-based, step, or target scaling.[98].

TABLE 1.2: OVERVIEW OF KEY AUTOSCALING SOLUTIONS OFFERED BY MAJOR PUBLIC CLOUD PROVIDERS [98]

Name	Providers	Description
Rule-based	*AWS, Azure*	A solution that relies on user-defined rules, triggering scaling actions when set conditions are met, such as when CPU utilization crosses a certain threshold.
Step	*AWS*	An evolution of rule-based solutions, enabling users to associate specific conditions with multiple scaling actions (or steps), thereby offering more fine-grained control over resource allocation.
Target	*AWS, GCP*	Advanced scaling solution that automatically maintains a target metric value by adding/removing instances without requiring users to write complex rules or policies.
Others	*AWS, Azure, GCP*	Proactive approaches that are complementary to rule-based, step, and target solutions. They work on larger time spans and may involve long-lasting pattern analysis of the workload.

1.6 Cloud workloads and workload patterns

Cloud workloads and workload patterns are related to each other, but they have some differences based on their focus and meaning.

1.6.1 Cloud workloads

Particular programs, services, or procedures that operate on cloud infrastructure are known as cloud workloads. The actual tasks or jobs being carried out in a cloud environment are represented by these workloads. In simpler words, the real things you

execute and manage in the cloud, such as programs or procedures, are called cloud workloads. The functional side of cloud operations is the main emphasis of the cloud workloads. Below are some of the examples.

- Hosting a website or web application (WordPress).
- Running a database service (PostgreSQL, MongoDB).
- Performing data analytics (Apache Spark jobs).
- Streaming services (video streaming platforms).
- Virtual desktops or hosted environments. (Azure virtual desktop, VMware Horizon)

Cloud-based workloads continue to grow very rapidly: there were over 504 million deployments in 2021, a steep 48% rise from the last two years. Software as a Service (SaaS) still stands at the top of available cloud models with a deployment share of 76%. The next contender is Infrastructure as a Service (IaaS) with a 15% share, followed by the remaining 9% of workloads being attributed to Platform as a Service (PaaS) [107].

Cloud workloads can be categorized into several sections based on their nature. Some of the examples are highlighted below.

- Classifying Workloads by Cloud Deployment Model. (IaaS, PaaS, SaaS)
- Classifying Workloads by Cloud-Native Technology. (VMs, Containers, Container as a Service (CaaS), Serverless)
- Classifying Workloads by Usage Patterns (Discussed in 1.6.2)
- Classifying Workloads by Resource Requirements (Standard compute workloads, High CPU workloads, High GPU workloads, High-performance computing (HPC) workloads, Storage-optimized workloads, Memory-intensive workloads)

1.6.2 Cloud workload patterns

Workload patterns characterize the behavior, traits, and demand trends of workloads over time and represent them abstractly. Cloud workload patterns focus on the operational and behavioral aspects of workloads across time. By modeling and simulating workload behavior, workload patterns facilitate the efficient design and testing of cloud infrastructure and autoscaling technologies.

Generally, Cloud computing workload can be categorized into five patterns [108]. These five patterns are static, high growth, on-off, periodic bursting, and aperiodic bursting. Additionally, to these basic workload patterns, in this thesis, we have discussed and tested five extra workload patterns which are derived by combining the original basic workload patterns. Namely, they are High growth - static - high shrink, High growth - static - high shrink On-Off with no Interval, High growth - static - high shrink On-Off with Interval, High growth On-Off, and High growth - High shrink. These workloads and their real-world applications are discussed in detail below.

1.6.2.1 Static workload

In this pattern, the volume of traffic is constant across time with little to no variation. There isn't much variation in the traffic.

Examples:

- Corporate blogs or websites: A business website that doesn't see significant surges in traffic or has a lot of dynamic material. Static material such as contact forms, informative pages, and basic resources could be offered.
- Static file hosting: A straightforward file storage solution with regular, limited access, such as a personal picture album or rudimentary portfolio website.

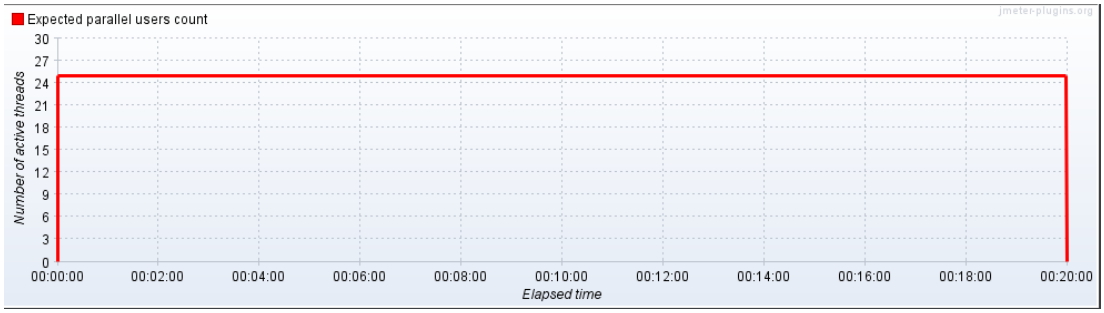

Fig 1.2: Static workload pattern

1.6.2.2 High growth workload

This trend entails steady traffic growth over time, frequently brought on by an expanding user base or more service utilization.

Examples:

- Social media platforms: An emerging social media platform that is becoming more and more well-known, as evidenced by the steady rise in user activity and traffic over time, such as Instagram or TikTok.
- SaaS platforms: As the user base grows, a SaaS company that is growing its features or customer base, such as Zoom or Salesforce, experiences steady, gradual traffic growth.

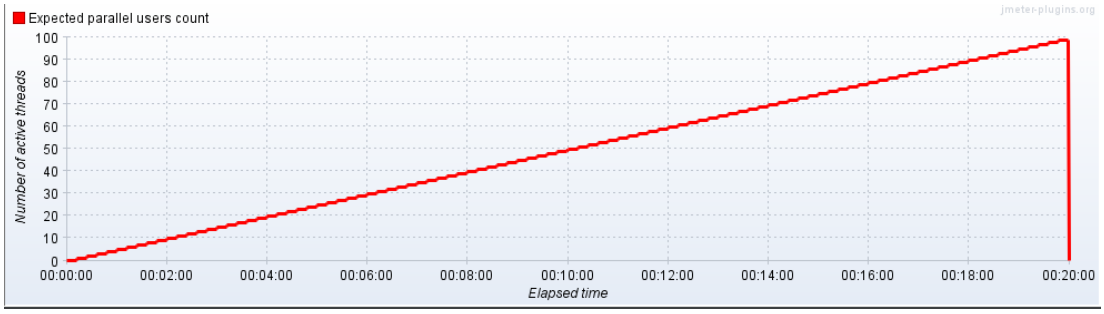

Fig 1.3: High growth workload pattern

1.6.2.3 On-Off workload pattern

Traffic that has clear boundaries and happens during separate on and off periods. For brief periods, the application might be required, but otherwise, it could be inactive.

Examples:

- E-commerce websites during promotions: When there is a flash sale or a brief promotion (like Amazon Prime Day or Daraz Black Friday), traffic explodes, but it eventually drops to zero or normal.
- Event-driven platforms: Websites that sell tickets, such as Ticket.lk or Kapruka.lk, may see a spike in traffic when they go on sale, followed by a sharp decline after the event.

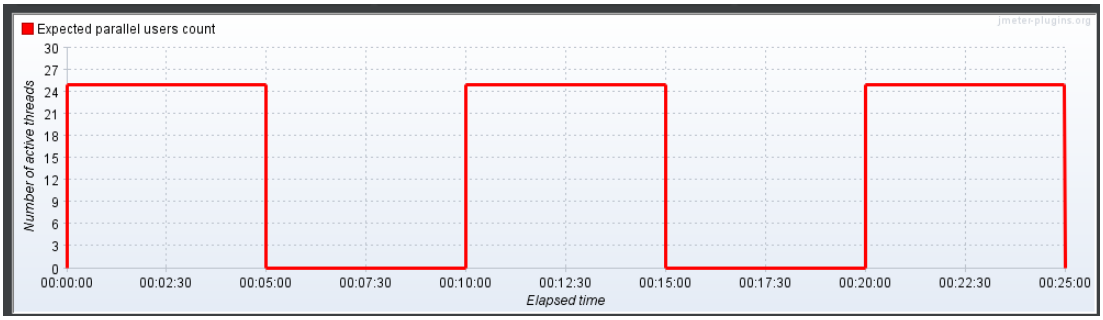

Fig 1.4: On-Off workload pattern

1.6.2.4 Aperiodic bursting workload pattern

Unpredictable, erratic traffic spurts that lack regularity or a set interval define this pattern. It might occur at any moment and without any discernible pattern.

Examples:

- News websites: CNN or the BBC when there are breaking news or viral stories that cause unexpected, abrupt spikes in traffic.
- Social media sites: When a post goes viral on sites like Twitter, Reddit, or Instagram, it might result in unexpected spikes in traffic that could happen at any time.

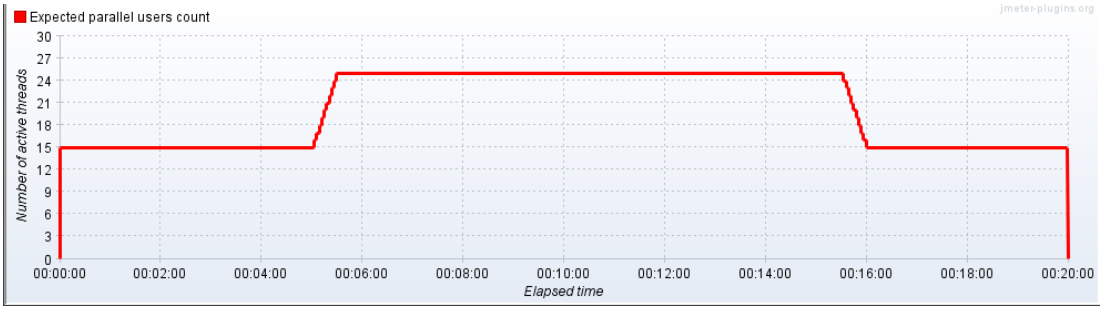

Fig 1.5: Aperiodic bursting workload pattern

1.6.2.5 Periodic bursting workload pattern

The load stays periodic rather than continuous, yet spikes in traffic at predictable times. These outbursts may occur on regular schedules (e.g., on specific days or hours of the day).

Examples:

- Retail websites: Online retailers like Amazon or Daraz that experience consistent spikes in traffic on specific days or at specific times of the day, such as weekends or Black Friday, or during specific times of the day, such as lunch or after work.
- News websites: Traffic increases during certain seasons of the year (e.g., New Year's Eve, elections) or during certain hours (e.g., daily news cycles or important events).

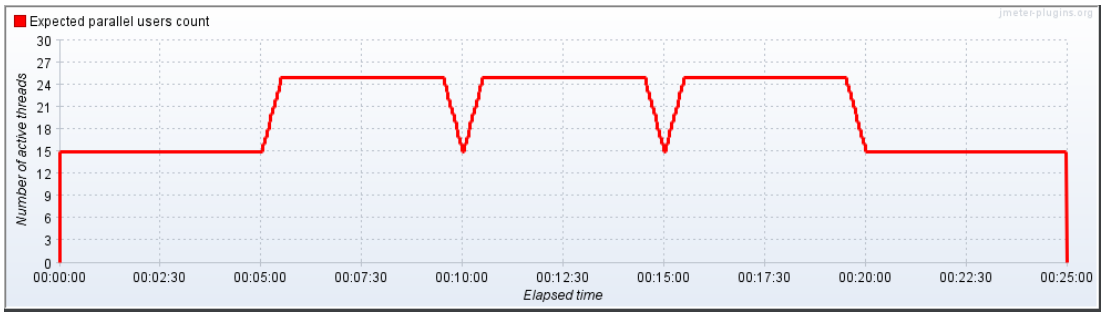

Fig 1.6: Periodic bursting workload pattern

1.6.2.6 High Growth - Static - High Shrink workload pattern

Traffic gradually increases before abruptly declining following a spike in activity. Following a period of expansion, the workload either reaches an impasse or returns to its previous level.

Examples:

- Tourism Websites: Websites such as srilanka.travel had a sharp increase in traffic during initiatives to promote travel (such as post-pandemic rehabilitation efforts or special events), but the traffic quickly declined once the vacation season concluded.
- Product launches: A new product from Tesla or Apple that generates a lot of excitement at first but then dies down after the initial launch period.

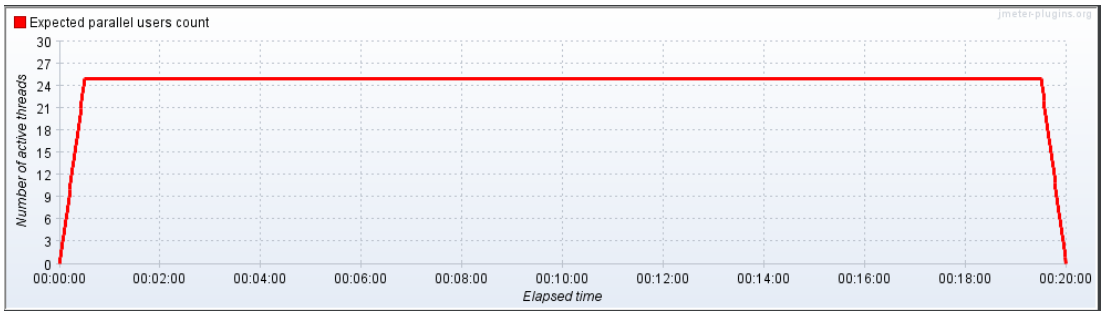

Fig 1.7: High growth-static-high shrink workload pattern

1.6.2.7 High Growth - Static - High Shrink On-Off with No Interval Workload Pattern

There is a sustained growth in traffic followed by sharp periods of on/off activity without a clear interval between the periods of activity and inactivity.

Examples:

- Sri Lanka Cricket: When people watch match footage or check scores in real-time during high-profile cricket matches, such as the Sri Lanka vs. India series, traffic increases significantly. However, as soon as the match is done, it rapidly decreases to almost nil. In subsequent games, this pattern recurs without a clear, regular break.
- Political events: A national election, for example, may see a sharp increase in website traffic during the campaign, but immediately after the election or when the event concludes, traffic immediately declines.

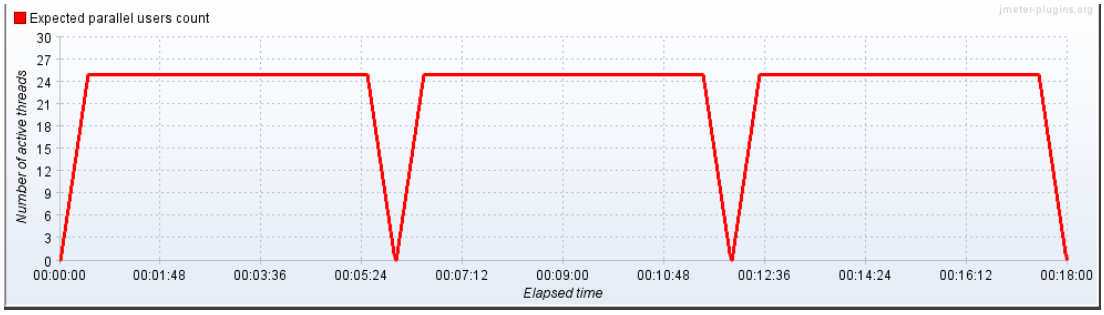

Fig 1.8: High growth-static-high shrink on-off with no interval workload pattern

1.6.2.8 High Growth - Static - High Shrink On-Off with Interval Workload Pattern

The on/off activity has a predetermined interval between each on-period, and each cycle repeats regularly until the traffic decreases. This pattern is similar to the preceding one.

Examples:

- TV Series: A series such as The Mandalorian or Game of Thrones. Over time, the program gains a sizable user following; traffic peaks around the premiere

of new seasons or episodes, and traffic declines between seasons (with regular on/off periods).

- Stock market trading platforms: There are reduced intervals of inactivity in between trading days and regular increases in activity during market opening hours.

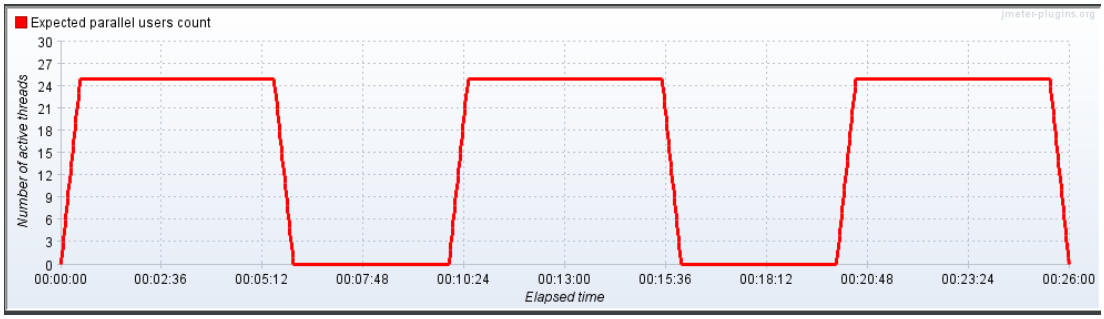

Fig 1.9: High growth-static-high shrink on-off with interval workload pattern

1.6.2.9 High Growth On-Off Workload Pattern

Over time, the traffic pattern experiences dramatic decreases (on-off) and abrupt bursts of high activity, although the rise is consistent.

Examples:

- E-commerce Launches: The online retailer Daraz.lk sees a spike in traffic during product launches or promotions (such as Flash Sales), but once the sale ends, traffic drastically declines before increasing again during the subsequent sale.
- Viral Marketing initiatives: When local businesses like Ceylon Tea or Dilmah run viral marketing initiatives, including influencer promotions or social media challenges, they may see unexpected spikes in traffic. After the advertising concludes, the traffic drastically declines.

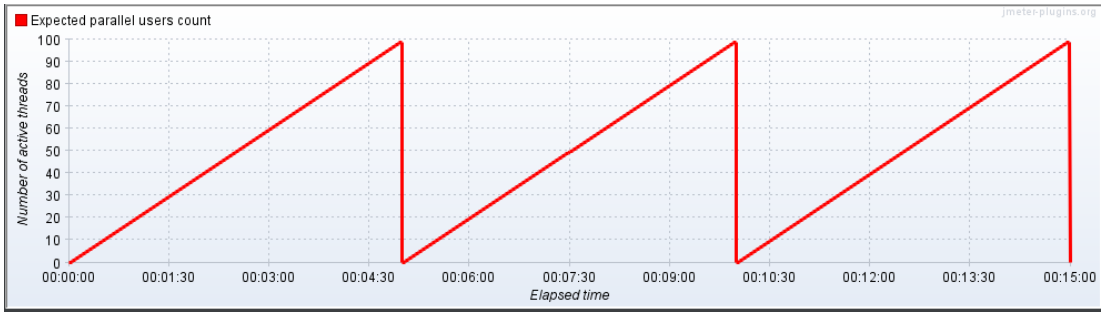

Fig 1.10: High growth on-off workload pattern

1.6.2.10 High Growth High Shrink Workload Pattern

A traffic pattern that experiences a fast surge followed by a severe decrease after initially growing greatly and then swiftly shrinking.

Examples:

- Mobile App Launches: Following an initial period of rapid increase in user sign-ups and engagement, a popular app such as Upay (a mobile payment app) or MyDoctor.lk (a telemedicine app) experiences a sharp fall. Traffic to the app may rapidly decline as its novelty wears off.
- Festivals & Promotional Offers: When Sri Lankan Airlines or nearby resorts offer special discounts during regional celebrations like Sinhala and Tamil New Year, there may be a significant spike in online traffic at first because of early reservations or promotions, followed by a steep drop after the festival season.

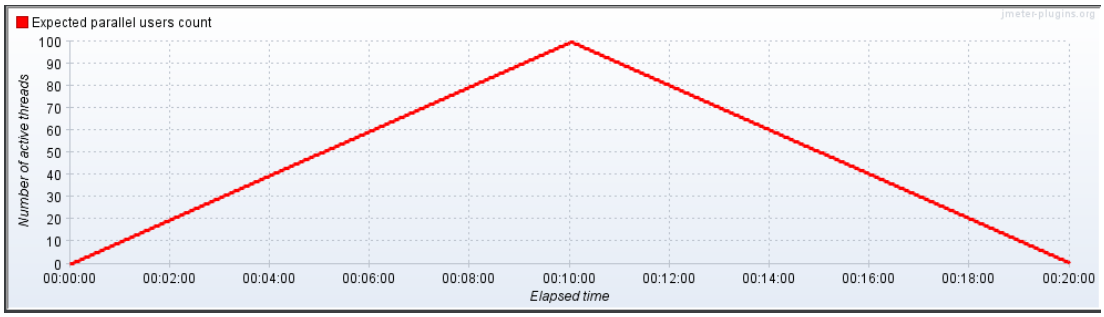

Fig 1.11: High growth High shrink workload pattern

1.7 OpenStack Services

OpenStack is a free-to-use open-source cloud computing platform that provides any solution to manage large pools of compute, storage, and networking resources in a data center. A further option of using these resources is through the web interface, Horizon, or programmatically by calling the OpenStack APIs. Commonly, OpenStack is implemented as an Infrastructure as a Service (IaaS) framework. OpenStack, with its attractive design and scalable principles, has a modular architecture; each component performs a specific function. These services interact and integrate through publicly accessible APIs, which are then used by the users to provision cloud services through these APIs as well as by internal components to coordinate operations.[12]. Below are the main components of OpenStack.

1.7.1 Dashboard (Horizon)

Horizon is an online graphical depiction designed for users to manage, provision, and interact with different OpenStack services. On the other hand, OpenStack APIs could be used by developers for creating or using tools for resource management purposes.[13].

1.7.2 Nova (Compute)

In essence, Nova constitutes the principal series of interfaces within an IaaS environment, provisioning and managing virtual machines (compute instances) on the fly. Nova can be thought of as a fabric controller for the cloud infrastructure, representing and manipulating compute resources on demand.[13].

1.7.3 Neutron (Network)

Neutron is the OpenStack networking service that creates connections and manages these connections among various OpenStack services, including compute. Neutron provides scalable, multi-tenant networking capabilities and APIs. Neutron supports both static and floating IPs to redirect traffic to enable maintenance or recovery from system faults. In addition, the modular and extensible architecture of Neutron permits a broad range of networking features, including firewalls and VPNs. [13].

1.7.4 Swift (Object Storage)

Swift is the OpenStack component that provides object storage for unstructured data. It operates as a scalable, distributed storage system accessible via APIs and is designed with high fault tolerance. Resilience is attained through redundancy by storing many copies of each file at different locations in the system. Upon hardware or server failure, OpenStack uses its internal logic to restore the lost data on a new node from another available source.[13].

1.7.5 Cinder (Block Storage)

Cinder supplies persistent block storage services for OpenStack compute instances. It allows users to dynamically create and manage storage volumes (Cinder volumes) using the native OpenStack API. These volumes can be provisioned on-demand without the user having to know the underlying hardware or physical location of the storage.[13].

1.7.6 Keystone (Identity)

The main responsibility of Keystone is to serve as the identity management system within OpenStack to handle authentication and authorization across all OpenStack services.[13].

1.7.7 Glance (Image service)

Responsible for storing and retrieving virtual machine (VM) disk images within Swift, OpenStack's object storage service, Glance deploys them to be directly accessible by OpenStack Compute through a RESTful API for new instance provisioning. Other components such as Heat may also interact with Glance to access image metadata and provide features such as high availability.[13].

1.7.8 Ceilometer (Telemetry)

Ceilometer is the metering and monitoring service of OpenStack, designed to collect, process, and analyze telemetry data from various OpenStack services and resources. It offers a unified framework for measuring usage data, performance metrics, and other relevant information across different cloud infrastructure components. By gathering and processing this data, Ceilometer enables administrators and users to gain insights into resource consumption, track system performance, and make informed decisions for optimization and planning.[63]

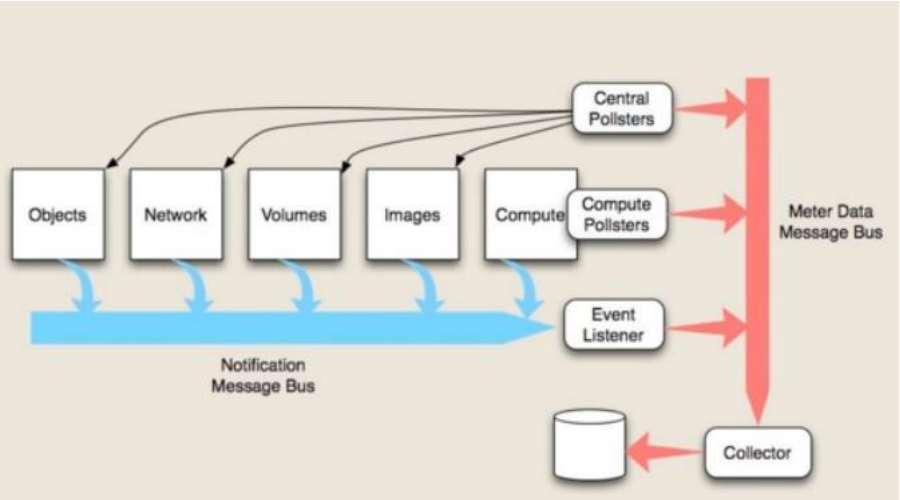

Fig. 1.12: Ceilometer design [63]

1.7.8.1 Key Features of Ceilometer:

Metering and Data Collection: Ceilometer collects data from various OpenStack services, such as Compute (Nova), Network (Neutron), Block Storage (Cinder), and Object Storage (Swift). It captures metrics related to resource utilization, including CPU usage, memory usage, network traffic, and storage capacity.

Metering Alarms: Ceilometer allows the creation of alarms based on specific metering thresholds. When a metering threshold is exceeded, an alarm is triggered, notifying administrators or triggering automated actions. This feature helps in proactive monitoring and enables timely responses to critical events.

Event Notifications: Ceilometer can also receive event notifications from different OpenStack services. These events provide important information about system activities, such as VM creation, deletion, or migration, network configuration changes, and storage operations. By analyzing these events, administrators can gain real-time visibility into the system's behavior and respond accordingly.

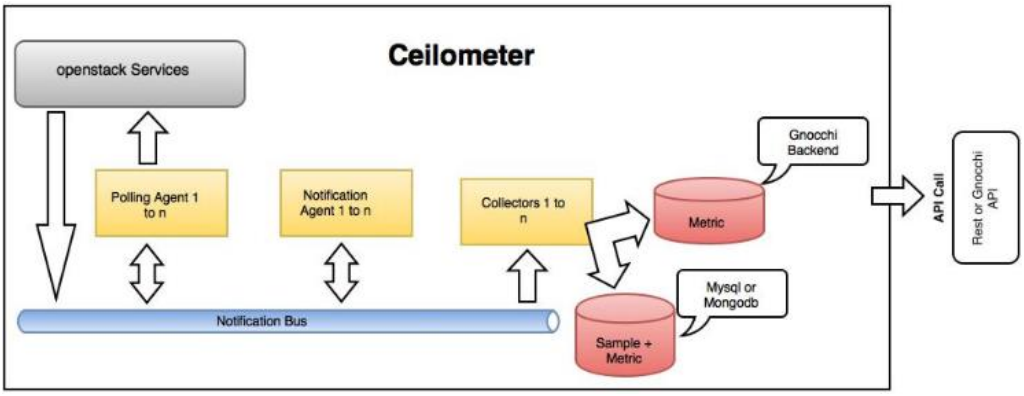

Fig. 1.13: Ceilometer event notification [63]

Data Processing and Storage: Ceilometer employs a flexible and scalable architecture for data processing and storage. It supports different data storage backends, including SQL and NOSQL databases. The data is processed using pipeline configurations, where it can be transformed, aggregated, and stored for further analysis.

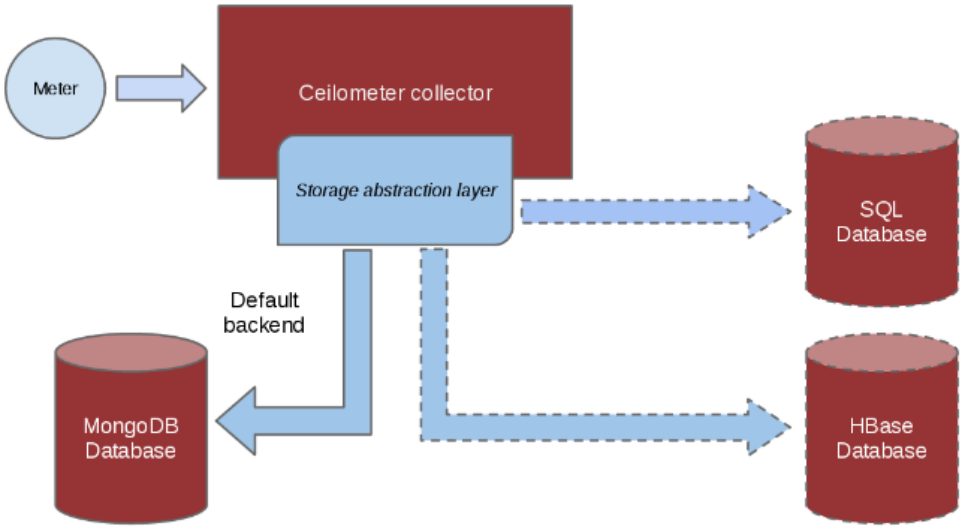

Fig. 1.14: Ceilometer data processing and storage [63]

Integration with Aodh and Gnocchi: Ceilometer integrates seamlessly with other OpenStack services like Aodh (Alarm service) and Gnocchi (Metric service). Aodh provides advanced alarm management capabilities, while Gnocchi offers a scalable and efficient metric storage and retrieval system. Together, these services enhance the monitoring and alerting capabilities of the Ceilometer.

APIs and User Interface: Ceilometer provides a RESTful API that allows users and administrators to interact with the telemetry data and perform various operations, such as querying meters, retrieving statistics, and managing alarms. Additionally, OpenStack's Horizon dashboard provides a user-friendly graphical interface for visualizing and analyzing telemetry data.

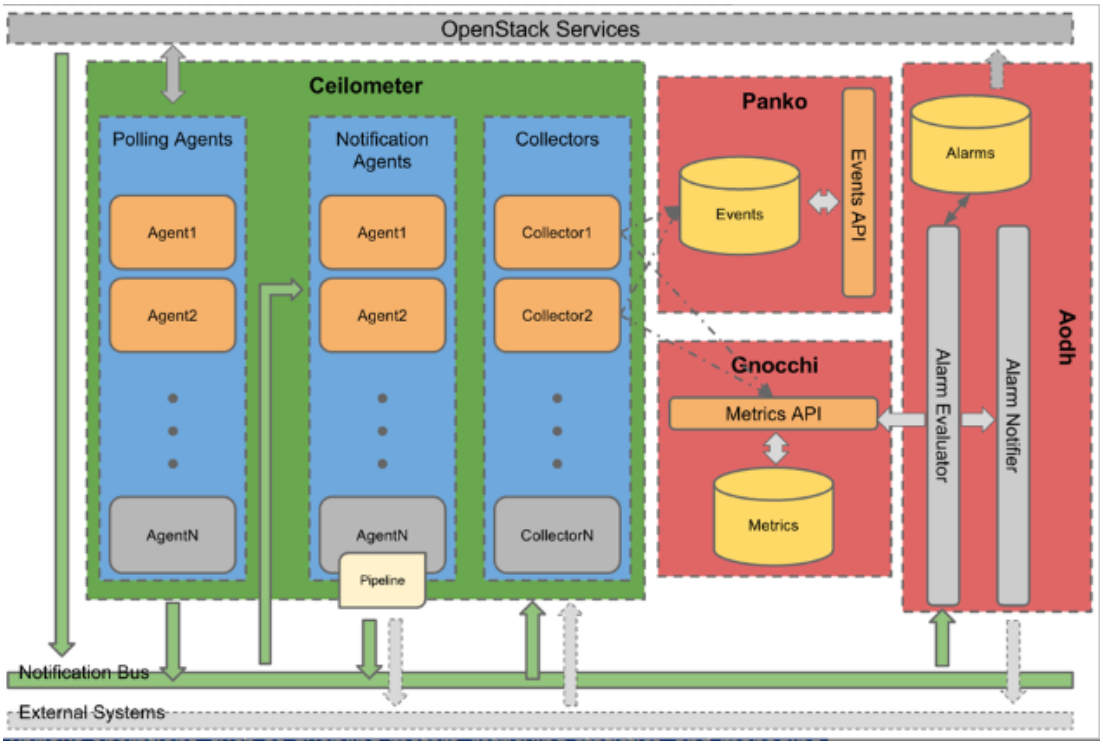

Fig. 1.15: Ceilometer high-level architecture [63]

1.7.8.2 Use Cases of Ceilometer:

Billing and Chargeback: Ceilometer's metering capabilities enable accurate measurement of resource consumption, making it a valuable tool for billing and chargeback purposes. Cloud service providers can generate detailed usage reports and invoices based on the metering data collected by Ceilometer.

Capacity Planning and Optimization: By monitoring resource utilization and performance metrics, Ceilometer helps in capacity planning and optimization. Administrators can identify underutilized or overutilized resources, detect performance bottlenecks, and make informed decisions for resource allocation and optimization.

Troubleshooting and Performance Analysis: Ceilometer's real-time monitoring and event notifications assist in troubleshooting system issues and analyzing performance. Administrators can track system events, monitor performance metrics, and correlate data to identify the root cause of problems and improve overall system performance.

SLA Management: Ceilometer's alarm and threshold capabilities enable service level agreement (SLA) management. By setting alarms based on predefined thresholds, administrators can ensure compliance with SLA commitments and take proactive actions to meet performance targets.

1.7.9 Aodh (Alarming)

Virtual Machines may have alarms set up to monitor specific thresholds like high memory use, weird shutdowns, and operational errors. OpenStack Aodh implements this functionality. Using Aodh, users can create, modify, and assign alarms to virtual instances. To track particular VM metrics, Aodh retrieves data from Gnocchi, the metric service. The OpenStack telemetry alarm component Aodh, fires alerts based on predefined rules to send alerts regarding monitored data under defined conditions. Alarms can exist in one of three possible states:

- ok - The condition being monitored has not been met
- alarm - The rule has been triggered based on current data.
- Insufficient data - There is not enough information available to properly evaluate the meaning of alarm status.

Aodh allows the creation of three types of alarms:

- Threshold Alarms: These alarms trigger an alert when a specified threshold is reached or surpassed. For example, a memory usage alarm would activate based on the memory consumption metric of a particular virtual machine (VM). The alarm will return to the ok state once the threshold is no longer exceeded.
- Event Alarms: These alarms are activated when a specific event occurs, such as an unexpected shutdown of a VM. Unlike threshold alarms, event alarms only move to the alarm state once and require manual intervention to reset their status.

Note: Event alarms will, by default, start in the insufficient data state the first time they are created. The state does not change until a certain condition specified in the event occurs.

- Composite Alarms: It evaluates multiple rules in an alarm as a group using logical operators to determine the alarm's state [64].

1.7.10 Gnocchi (Time series database)

Gnocchi is a time-series database designed to handle data from the telemetry service Ceilometer. It links measurements collected by Ceilometer to specific resources and metrics [64]. As a time series database, Gnocchi supports multi-project capabilities and stores metrics and resource data. Its purpose is to manage large volumes of metrics while ensuring that operators and users can access resource and metric information at scale [65]. Gnocchi addresses the challenge of efficiently storing and indexing time-series data for vast and dynamic systems, such as modern cloud platforms that are both large and multi-tenant [66].

It is very capable of taking care of a huge bulk of aggregated data in a high performing, scalable, and fault-tolerant manner and should avoid dependency on very complex storage systems from the very beginning of its design. Unlike most time series

databases, Gnocchi stores pre-aggregated data before its storage. In fact, aggregation is normally provided as an optional feature that can be calculated at execution time during the query (averages, minimums, etc.). Instead, this method identifies Gnocchi in the time series storage area because it uses pre-aggregated results after ingesting data [66].

Metrics:

A metric is a quantifiable characteristic of a resource, be it memory usage, CPU time consumption, or number of vCPUs in use. The following are characteristics that define metrics:

- Name: The metric is uniquely identified by this name.
- Unit: This specifies the unit of measurement, e.g., MB, ns, or B/s.
- Resource ID: This is an identifier of the resource from which the measurement is obtained.
- Unique User ID (UIID): This is a unique metric specific identifier.
- Archive Policy: This dictates the aggregation and storage of measurements for the metric. Each metric for every resource is linked with the archiving policy, which defines for how long and how many data points should be collected and how they should be aggregated could be used.

In OpenStack, Ceilometer automatically populates Gnocchi with metrics and resources. In Gnocchi, granularity refers to the time interval between each aggregated data point [64].

1.7.11 Octavia (LBaaS)

Open source load-balancing infrastructure intended to function at scale and fit seamlessly into OpenStack's ecosystem, Octavia delivers load balancing through the orchestration of pools of virtual machines or containers, or indeed bare-metal servers, called amphorae, which are created on demand. What distinguishes Octavia from traditional load balancing methods is its unique cloud native attributes, such as quick and flexible horizontal scaling that works on demand. Load balancing in a cloud environment is an extremely important consideration for delivering applications with great scalability and the highest availability and is, therefore, a core function of the cloud. For this reason, Load Balancing as a service is considered the infrastructure of clouds, while Octavia, like Nova, Neutron, and Glance, is considered to be equally primary to OpenStack. To carry out the above functions, Octavia interoperates with various other components within the OpenStack ecosystem.

- **Nova -** Takes care of the orchestration of the lifecycle of amphorae by provision of computing resources dynamically when required.
- **Neutron -** Links amphorae to tenant networks and external environments for network connectivity.

- **Barbican** - Used for management of TLS certificates and credentials when termination of the TLS is enabled on amphorae.
- **Keystone** - The resource for authentication of users accessing the API of Octavia and makes the secure interaction of Octavia with the other services of OpenStack possible.
- **Glance** - Hosts the image of the virtual machine that deploys the amphorae.
- **Oslo** - Internal communication is supported among the components of Octavia's controller, in line with OpenStack's coding standards, review processes, and project architecture.
- **Taskflow** - Part of Oslo (but significant), used by Octavia for managing backend service setups and operational workflows.

Octavia consists of a number of component parts, including:

- **Amphorae** - Any of the separate virtual machines, containers, or bare metal servers that provide load balancing for tenant application environments is referred to as an amphora. The standard amphorae for Octavia version 0.8 is an Ubuntu virtual machine with HAProxy inside.
- **Controller** - The controller is the central management unit of Octavia. It consists of five subcomponents, each represented by its own daemon. These daemons can be deployed on separate back-end infrastructures, if needed:
    - **API Controller** - This, as the name suggests, manages the API of Octavia. It takes the request from the API and sanitizes it somewhat before forwarding it to the controller worker by way of the Oslo messaging bus.
    - **Controller Worker** - This performs the function of processing the cleaned API commands received from the API controller and actually does the actions that are required to fulfill the API request.
    - **Health Manager** - This part constantly evaluates the operational status of each amphora to confirm that it is functioning properly, including managing failover processes when any amphora goes down.
    - **Housekeeping Manager** - Carries out the clearing of old entries from the database and supervises the rotation of certificates with which the amphorae are used.
    - **Driver Agent** - The driver agent is responsible for gathering and sending status reports and statistical information from the provider drivers to ensure normal communication between components.
- **Network** - The entire usefulness of Octavia is the manipulation of its network configurations. Amphorae are deployed with network interfaces up on the load balancer network and may then be connected to tenant ones directly through the load balancing configuration set by the tenant to reach the backend pool

members, depending on the specific load balancing configuration set by the tenant.[67].

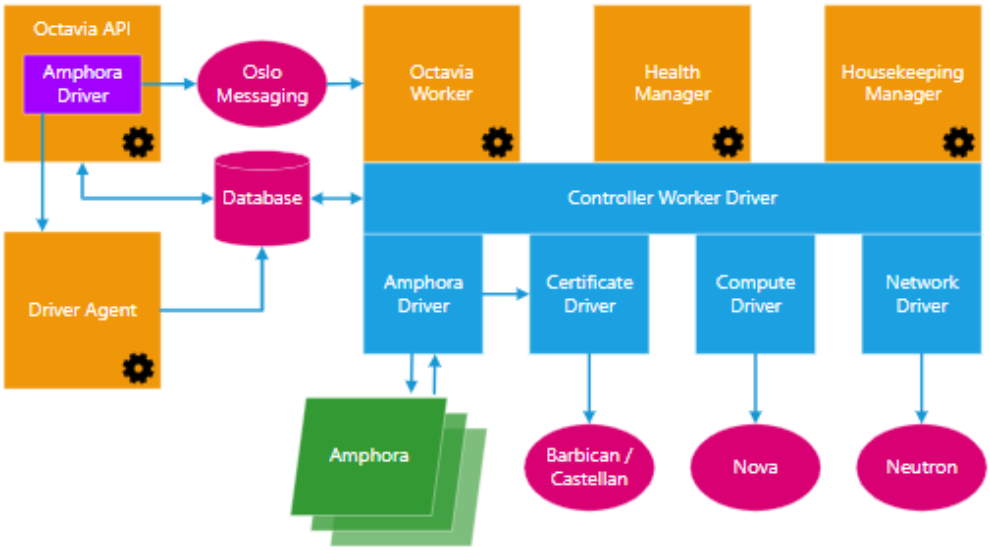

Fig. 1.16: Octavia architecture [67]

1.7.12 Heat (Orchestration)

Utilizing the native HOT template format or the AWS CloudFormation template format, Heat controls the deployment of complex cloud applications. It automates the creation of a functional stack, including compute instances, storage, and floating IPs, based on a template defined in a text file [79]. Heat is the main project that is in charge of OpenStack services to keep them in sync, created on 23 May 2013. The primary aim of Heat is to manage the full lifecycle of an application and infrastructure by providing human readable services within OpenStack clouds. Although Heat has been or will be chiefly for infrastructure management, it is also considered to manage software using templates; configuration management tools like Puppet and Ansible are often used in doing such tasks. Besides, any add-in that can be installed, according to the appropriate plug-in, enhances the flexibility of Heat by many plugins, at the same time allowing it to be customized.[68].

1.7.12.1 How it works:

- We could regard a Heat template as a human-readable text file describing the infrastructure for a cloud application, which lends to its suitability with version control, comparisons, and modifications.
- It essentially serves as a description of the various infrastructure resources, such as servers, floating IPs, volumes, security groups, users, etc.
- Heat includes an autoscaling service that works with Telemetry and allows scaling groups to be included as a resource within the template.

- Templates also define interrelations between the resources (for example, associating a volume with a server); this allows the Heat system to call the OpenStack APIs to provision infrastructure in the proper order for full application deployment.
- All application lifecycle management, allowing for easy updates by only editing the template, which Heat will use to make any required changes. To ensure that resources are cleaned up when the application is no longer needed.
- Heat focuses mainly on managing the infrastructure that can readily interoperate with software configuration management tools such as Puppet and Chef [69].

1.7.12.2 Terminology Encountered while Using Heat Service:

- **Resources:** These consist of the parts of networks, volumes, subnets, and security groups. Orchestration involves creating, modifying, or managing these resources.
- **Stack:** A stack is a set of resources grouped together.
- **Parameters:** With parameters, inputs can be fed into the template at deployment time, and these can be changed at runtime if needed.
- **Template:** A Template is a file that defines the format and configuration of stack resources with code.
- **Output:** The output typically displays processed information or results to user regarding the status or details of the resources deployed.

Primary Components of Heat Architecture:

Heat architecture involves four main components that serve different purposes:

- **Heat:** This Command Line Interface (CLI) is used for client access to heat API. The tool is not an obligation since it's possible to directly connect to heat API..
- **Heat-API:** Processes and forwards incoming requests to the heat engine for execution through the OpenStack native REST API.
- **Heat-API-cfn:** A query interface based on AWS CloudFormation that translates API requests into those understood by the heat engine.
- **Heat-engine:** The Heat engine is the very heart of Heat, orchestrating services with the launch of templates, also causing an event to be dispatched to the API users.

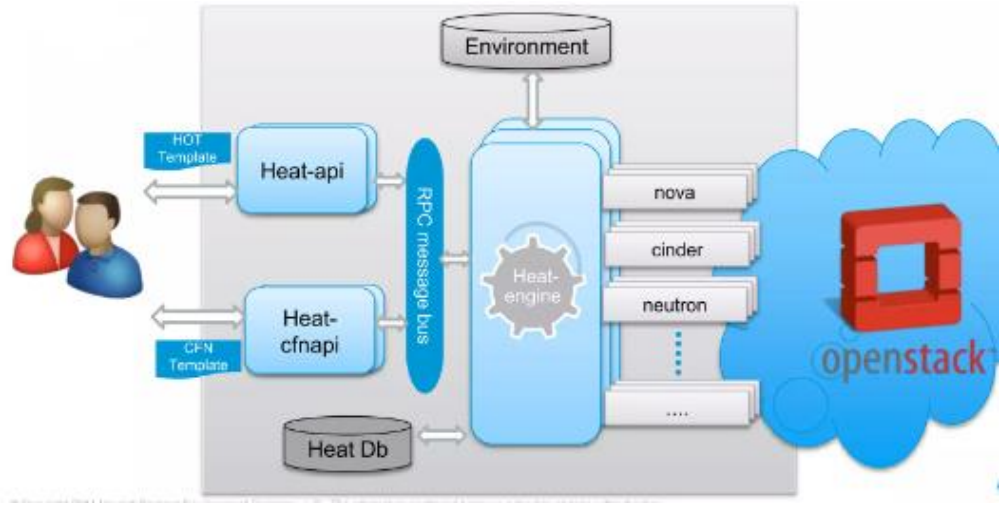

Fig. 1.17: Heat architecture [70]

1.7.13 Senlin (Clustering)

Senlin is a service that OpenStack offers for clustering. This is used to create and manage clusters of identical resources from various OpenStack services. The main purpose of Senlin is to easily orchestrate collections of similar objects. On the whole, the creation and operation of resource clusters by Senlin allows even greater management of the resource grouping within the cloud environment through interaction with other OpenStack services. For the majority part, Senlin will handle its interaction with other OpenStack services through profile plugins. Each type of profile provides the way in which Senlin manages the lifecycle of a particular resource type, such as for creating, updating, or deleting it. Clusters in Senlin can also be associated with different policy objects. Policies define the behavior of the cluster and can vary in terms of how stringent the enforcement is. Users can utilize service APIs to manage clusters freely through adding or removing nodes on the fly and using or not using several policies, creation, deletion, scaling, load balancing, and health monitoring. The tightness of integration between the other OpenStack components means that Senlin's deployment and orchestration capability can streamline large-scale resource environments. Its architecture is extensible enough to support managing a seemingly endless number of object types, while each object's complete lifecycle is governed by a corresponding profile type. These profile types are implemented as plugins, which the service engine can load dynamically to perform resource-specific actions.[71].

1.7.13.1 Components of Senlin:

Senlin is a command-line interface (CLI) utility meant to interact with the Senlin API in order to manage various elements of the cluster, such as nodes, profiles, policies, actions, and events. For developers, Senlin also provides access to a RESTful interface.

Senlin-api offers an OpenStack standard compliant RESTful API, which processes the incoming API requests and passes them to the Senlin engine via RPC..

Senlin-engine is responsible for the orchestration of clusters, nodes, profiles, and policies that define how resources will need to be treated over time.[72].

Key features of Senlin:

- Flexibility is built into the cluster framework, intended for handling homogeneous collections of cloud resources within the OpenStack ecosystem.
- A range of APIs for cluster membership operations, such as the addition and removal of nodes.
- It provides an extensible profile management system based on plugins to support a large variety of resource types.
- Policy enforcement framework supporting user defined plugin-based policies regarding cluster behavior or governance.
- A plugin-based event system, which can be used either for logging or to integrate downstream tools for further processing.
- An asynchronous task execution model that preserves consistent and well-formed states across a cluster and its constituent nodes.

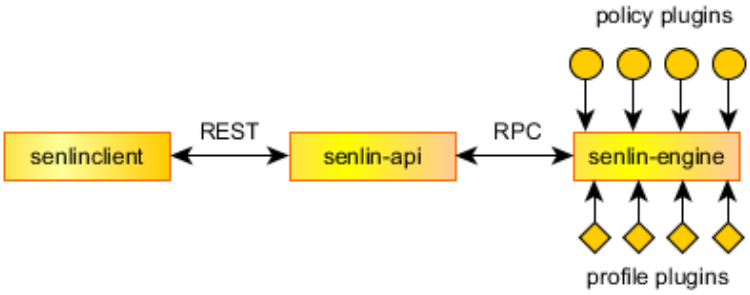

Fig. 1.18: Senlin architecture [71]

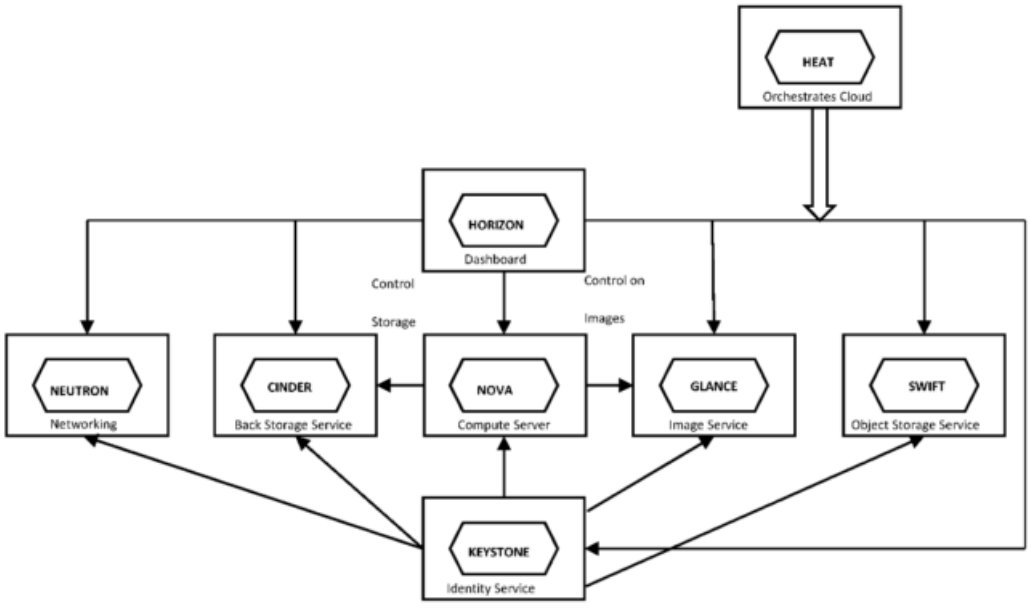

Fig. 1.19: OpenStack system architecture [62]

1.8 Theory of Autoscaling in OpenStack

The autonomous recognition of workload changes and self-adjusting operation of the system without the intervention of a manual cloud operator is called autoscaling in OpenStack. Auto-scaling includes, but is not limited to, the provisioning and scaling of the control plane and other cloud resources and is most commonly understood in terms of scaling compute workloads. These automatic functions could comprise both scaling out (which increases capacity) and scaling in (reducing capacity) operations with the intention of optimizing resource allocation. The ultimate benefit of this duality is to maximize user experience while at the same time optimizing costs and operating efficiency [73].

Auto-scaling can be implemented through various mechanisms provided by OpenStack in a cloud environment, such as Heat AutoScalingGroup or Senlin clustering service. However, difficulties in achieving automated scaling are still faced by users and system operators. The integration of multiple services causes fragmentation in the user experience because, normally, OpenStack development is directed towards individual components. OpenStack, very much, has a lot of essential building blocks for autoscaling, but there is still a need for a more unified and user-friendly approach to bringing collaboration towards development and avoiding redundant implementation efforts. [73].

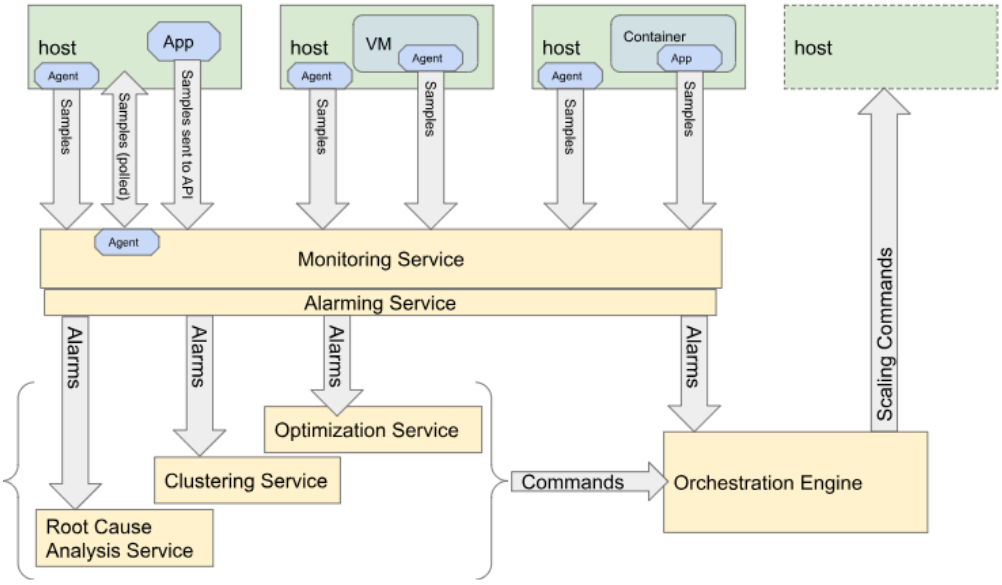

Fig. 1.20: OpenStack autoscaling conceptual diagram [73]

1.8.1 The main components of autoscaling:

OpenStack offers a complete package of services regarding the definition and management of all or parts of a cloud environment, including orchestration and provisioning of cloud services. It provides cloud administrators to customize infrastructure as per specific user requirements.

Auto-Scaling Targets – Within the OpenStack ecosystem, auto-scaling can be applied to various infrastructure components, including:

- Compute Nodes
- Virtual machines operating on compute nodes
- Containers deployed on the compute infrastructure
- Network-attached storage resources
- Virtual network functions (VNFs)

Monitoring - Ceilometer from the Telemetry project, Monasca

Alarming - Aodh from the Telemetry project, Monasca

Orchestration Engines – Heat, Senlin

1.9 Need for auto-scaling in private clouds

Auto-scaling in private clouds is a strategic capability that enables organizations to efficiently manage resources, enhance performance, and ensure optimal user experiences while controlling costs. It aligns with the dynamic and evolving nature of modern IT infrastructures, providing a scalable and responsive solution for private cloud environments. Below are the key reasons highlighting the need for auto-scaling in private clouds.

1.9.1 Dynamic Workloads

Workloads on a private cloud can vary based on factors like time of day, business cycles, and user demand. Auto-scaling allows the infrastructure to dynamically adapt to changing workloads by provisioning or de-provisioning resources in real time.

1.9.2 Cost Efficiency

Provisioning resources to handle peak loads always can lead to underutilization during periods of lower demand. Auto-scaling optimizes resource usage, enabling organizations to allocate resources based on actual demand. This leads to cost savings by avoiding the continuous operation of unnecessary infrastructure.

1.9.3 Improved performance

The inability to handle sudden spikes in traffic or demand can result in performance degradation and service disruptions. Auto-scaling ensures that additional resources are automatically added during peak demand, maintaining optimal performance, and preventing overloads.

1.9.4 Resource Optimization

Static resource allocation may lead to inefficient utilization of computing, storage, and network resources. Auto-scaling enhances resource efficiency by automatically

modifying the number of virtual machines or instances in response to fluctuating demand, thereby ensuring optimal utilization of infrastructure resources.

1.9.5 Enhanced Availability and Reliability

Traditional static environments may struggle to maintain high availability during unexpected events or traffic spikes. Auto-scaling enhances availability by distributing workloads across multiple instances and automatically replacing failed instances. This improves reliability and reduces downtime.

1.9.6 Efficient Resource Planning

Without accurate visibility into future demand, it's challenging to plan resources effectively. Auto-scaling relies on monitoring and metrics to predict and respond to changes in demand, enabling more informed resource planning and allocation.

1.9.7 Quick Response to Unforeseen Events

Rapid changes in user behavior or unexpected events can lead to sudden increases in demand. Auto-scaling responds promptly to changes, automatically adding or removing resources to maintain optimal performance and responsiveness.

1.10 Monitoring and visualizing OpenStack resources

In this project, for the purpose of monitoring and visualization of resources which created on the OpenStack cloud, Prometheus, Grafana, and OpenStack-exporter are used.

1.10.1 Prometheus

Prometheus is a powerful open-source monitoring system written in Go and designed for the purpose of collecting and storing time series metrics for extended periods of time. It was initially developed at SoundCloud in 2012 and was later donated to the Cloud Native Computing Foundation (CNCF) in 2016. Over the years, Prometheus has become a core element in the cloud native observability toolkit. It has a multidimensional data model and its own query language, PromQL, allowing for complex querying on the metric data. Metrics are collected by scraping HTTP endpoints at defined intervals, thereby providing high-resolution monitoring across distributed systems. Now let’s discuss how Prometheus is working,

1. Data Collection and Retrieval in Prometheus: Metrics are collected in a pull-style fashion; Prometheus scrapes the relevant targets for metrics as per a defined schedule. Targets are generally applications, services, or infrastructure items that must be instrumented using a Prometheus client library to expose relevant metrics via HTTP endpoints. In this way, Prometheus efficiently pulls in time-series data for monitoring and analytical assessment.

2. Data Storage in Prometheus: Prometheus keeps the metrics it acquires stored in its time series DB, which simply stores time-stamped data points under some metric label. Such a structure makes it easier to preserve and analyze system performance trends over a long period: live monitoring as well as geographically remote back study of infrastructure behavior over a set period of time.

3. Service Discovery in Prometheus: Prometheus uses varying dynamic service discovery methods to automatically detect and bind new service instances to its monitoring scope. This makes manual configuration unnecessary, ensuring that observability remains ongoing within fast-evolving distributed environments.

Now let’s discuss the main options of Prometheus:

- Multi-Dimensional Data Representation: A flexible data model has been taken as Prometheus's foundation so that it can support multi-dimensional time-series data. Metrics are labeled with relevant tags, such as the job name and instance identifier, which allow for fine filtering as well as aggregation, giving the user an opportunity to acquire deep insights into system behavior.
- PromQL (Prometheus Query Language): Among all these, Prometheus has its own language for querying, that is PromQL. It allows appending the maximum expressiveness and efficiency in querying time-series data. This feature further enhances the capability of performing comprehensive monitoring and performance evaluation in heterogeneous infrastructures.
- Rule-Based Alerting: Alerts can be incorporated using rules in Prometheus against a given condition or threshold. When any condition is met, then an alert is triggered, which enables early warning in case an anomaly or degradation happens in the system, to that users will not feel the impact of the event.
- Visualization through External Tools: In most cases, Prometheus has a built-in integration with visualization platforms such as Grafana for creating dynamic and engaging dashboards that add more meaning to the metrics collected and can be useful for monitoring the system as well as its reporting.[74].

1.10.2 Grafana

Grafana is an open-source analytics and monitoring application for the purposes of monitoring and visualizing large scale system metrics and application performance indicators on dashboards that can be customized in various ways. It has a rich ecosystem built around data source integrations ranging from Prometheus, Graphite, InfluxDB, Elasticsearch, MySQL, to even PostgreSQL. From these sources, Grafana can either pull real-time or historical data to either interpret or explore several complex datasets, often seen in a distributed cloud environment. There are a number of panels on the dashboards, each with a specific nature of visualization, such as time series charts, histograms, heat maps, or geographic maps. These visualization types enable organizations to draw actionable insights and make decisions based on observed patterns and anomalies. Each panel operates independently, providing a separate view

on particular metrics or the KPIs of different components of the monitored infrastructure.[75].

Fig. 1.21: Grafana dashboard [75]

1.10.3 OpenStack-exporter

The OpenStack-exporter is a tool used for integrating OpenStack metrics with monitoring systems, particularly Prometheus. It functions as a bridge to expose OpenStack metrics in a format that Prometheus can scrape and store for analysis. Below is a high-level overview of this service.

1. Metrics Collection: It gathers various performance and operational metrics from different OpenStack services (e.g., Nova, Neutron, Cinder).

2. Metrics Export: The gathered metrics are then exposed via an HTTP endpoint in a format that Prometheus understands.

3. Monitoring Integration: Prometheus can scrape this endpoint at regular intervals to collect and store the metrics. This data can then be visualized using tools like Grafana or used to set up alerts.

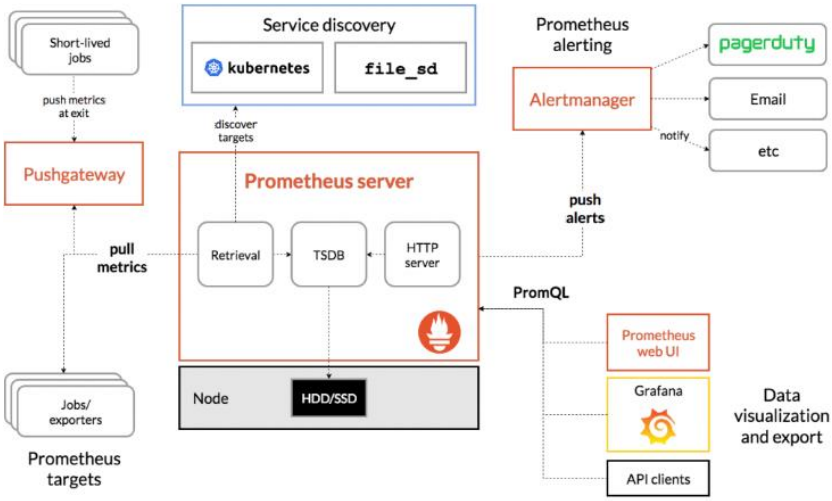

Fig. 1.22: Prometheus Architecture [76]

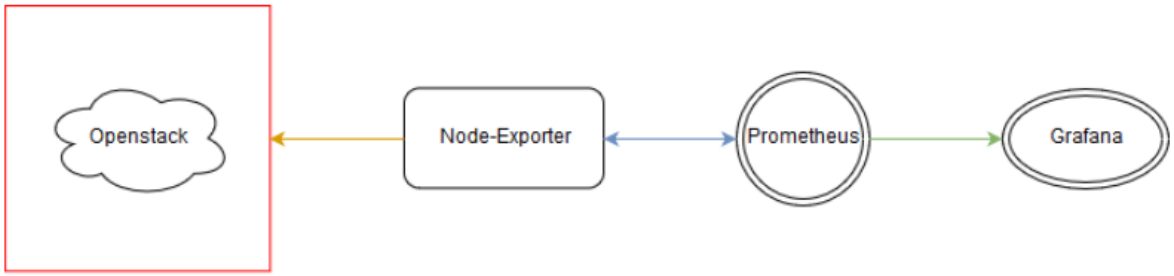

Fig 1.23: Monitoring setup used in this project [77]

CHAPTER 2

RESEARCH PROBLEM

As cloud computing continues to grow in popularity, scalability and effective resource management are now essential for businesses. Numerous methods are available for controlling autoscaling, a crucial function that dynamically modifies computer resources to fit workload needs, in OpenStack, a popular open-source cloud platform. Senlin, the clustering and policy-based autoscaling service, and Heat, the orchestration service, are OpenStack's two main autoscaling engines. Both engines provide distinct methodologies and features for managing resources; however, their comparative performance, scalability, and efficiency under varying workloads remain underexplored in existing literature.

Heat is a general-purpose orchestration engine that defines resource scaling strategies using templates. Although adaptable, its event-driven design could have drawbacks in situations with complicated dependencies or frequent scaling events. Senlin, on the other hand, provides a more specialized method of autoscaling with the introduction of policy-based scaling, fine-grained control over resource behavior, and clustering capabilities. Despite these developments, further research is needed to determine how well it performs in the real world while managing dynamic and varied workloads in comparison to Heat.

In this situation, choosing the best autoscaling engine for their unique requirements might be difficult for enterprises. This selection is heavily influenced by factors including response speed, scalability, workload characteristics, and simplicity of configuration. Furthermore, academics and practitioners looking to improve their cloud infrastructures are left in the dark by the absence of thorough benchmarks and consistent comparisons between Senlin and Heat.

This research aims to address the gap by systematically evaluating the autoscaling performance of Senlin and Heat using synthetic workloads generated from Apache JMeter. The study seeks to provide insights into their operational differences, scalability, and efficiency, enabling stakeholders to make informed decisions regarding their use in production environments. By doing so, this research contributes to the broader understanding of autoscaling mechanisms in OpenStack and their applicability to real-world scenarios.

CHAPTER 3

RESEARCH OBJECTIVES

The following are the main goals of this study, which address the research problem specified in the research problem section.

- To review the existing analyses done by the other researchers on comparing autoscaling solutions in the cloud computing domain.
- To review the existing OpenStack-based and non-OpenStack-based auto-scaling solutions, and to find their algorithms, their communication topology among servers, and their load balancing (work distribution) methods.
- To review different types of auto-scaling algorithms and workloads in the cloud environment.
- To Establish a private cloud infrastructure using OpenStack.
- To implement the auto-scaling engines Heat and Senlin on the OpenStack.
- To generate synthetic workload patterns like the real workloads using Apache JMeter.
- To conduct comprehensive testing on the Heat and the Senlin autoscaling engines with the help of the above-mentioned workloads and document the entire process including findings, test cases, and results.
- To analyze the results using the MCDA (Multiple Criteria Decision Analysis) evaluation technique and come up with the optimal auto-scaling solution for each kind of workload pattern.

CHAPTER 4

LITERATURE REVIEW

In this section, we will review and evaluate the work that other researchers have done related to analyzing and comparing the autoscaling solutions in the domain of cloud computing, also reviewing the existing autoscaling solutions in OpenStack and other Private and public cloud providers. A direct exploration of analyzing and comparing the autoscaling solutions in OpenStack with a combination of implementations yielded limited results in the literature. However, an extensive body of work addresses the topic of autoscaling in broader cloud environments, including OpenStack. The dynamic nature of the demand for autoscaling solutions enhances the importance of exploring advancements in this field. Autoscaling is essential for satisfying the dynamic demands of digital services, and it is critical to comprehend the current environment to suggest novel approaches. While there isn't much research specifically on the combination of implementation, comparison, and analysis of the autoscaling engines on OpenStack, we will be able to better understand this specialty by utilizing knowledge from related studies. Our objective is to gather knowledge from prior studies, spot gaps in the field, and provide the groundwork for evaluating the Heat and the Senlin, which are the two autoscaling engines in OpenStack.

4.1 Studies related to analyzing and comparing the autoscaling solutions in the domain of cloud

In a study conducted in 2024 [98], the researchers said to present a comparative analysis of auto scaling mechanisms provided by three leading public cloud services: Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure. Besides evaluating these commercial solutions, they have benchmarked them against two self-created schemes, ScaleX, which is based on its design principles from control theory, and QN-CTRL, which came from queuing theory-based solutions. The evaluation of the different auto-scaling strategies was carried out using a combination of a custom-built simulation framework and actual cloud-based deployments. These setups were subjected to a range of synthetic and real-world workload traces to thoroughly assess system behavior. From the overall analysis that studied simulated outputs and empirical measurements, ScaleX and QN-CTRL were shown to dominate commercial auto-scaling techniques in general. They achieve a better trade-off between minimizing service-level agreement (SLA) violations and optimizing resource utilization than most test scenarios.. In their work, they considered 8 numbers of autoscaling systems including their solutions and solutions that are available in well-known public clouds GCP, AWS, and AZURE. Those autoscaling systems are mentioned below.

1. Static autoscaling (1) - This functionality is available in AWS, GCP, and Azure.

2. Rule-based autoscaling (1) - Configurable rule-based auto-scaling is realized by AWS and Azure. The authors implemented specific scaling rules based on response time under the service level agreement (SLA). If the response time exceeds 90% of the SLA threshold, one CPU core is added; if the response time falls below 50% of the SLA, one core is removed.
3. Rule-Based Autoscaling (+3) Supported in AWS and Azure: In this strategy, the autoscaling mechanism has predefined threshold-based rules, like ones used in basic scaling configurations. However, instead of adjusting by a single core, the system increases or decreases the number of CPU cores by three when the specified conditions are met (e.g., response time significantly deviating from the SLA target). This approach enables more aggressive scaling responses to workload fluctuations.
4. Step-Based Autoscaling Supported in AWS: AWS supports stepwise autoscaling. It relies on a hierarchical policy adjusting resources as per the degree of deviation in SLA based response times. For instance, when response time crosses 130% of SLA thresholds, scaling increases the number of cores by 30%. Between SLA and 110% of the SLA threshold, the upscaling comprises 20%. Brackets of response times from 90 to 100% SLA yield a 10% increase. When response time drops to less than 80% of the SLA, scaling down means reducing the core by 10%. This aids in fine control and gradual scaling response concerning variations in workload.
5. Target-Based Autoscaling Supported in AWS and GCP: In this method, the Horizontal Pod Autoscaler (HPA) in Kubernetes works to keep a performance metric, say CPU utilization close to a defined target. Instead of using fixed rules, it automatically increases or decreases the number of pods in real time based on demand. This makes resource management easier and more flexible without needing manual scaling settings.
6. Target fast autoscaling - This method extends the target-based autoscaling setup by implementing a shorter control interval, enabling quicker adjustments to workload changes.
7. ScaleX – This is their custom autoscaling solution, which is designed using control theory.
8. QN-CTRL - Their solution which is designed using queuing theory.

These synthetic workloads are utilized for experimentation and evaluation purposes in control systems during their design phase, reflecting the real-world attributes of such real workloads in an abstract way. Load intensity can be anything as follows, depending on what type of workload is being modeled; expressed by the number of requests that have to be processed at any given time, req(t). Each of these workloads is described below.

1. Sine (SN1, SN2) – This is a repetitive workload with a sinusoidal pattern: smooth fluctuation over time.
2. Stari (ST1, ST2) – This workload has made abrupt switching between two intensity levels (for example, low and high) in a measure that more closely resembles an impulse.
3. Ramp (RP1, RP2) – This workload increases or decreases steadily in a linear fashion until it reaches a specified value, after which it stabilizes.
4. Twitter (TW) – This is a real-world dataset that records the number of tweets posted every second on January 1st, 2021.

TABLE 4.1- WORKLOADS USED TO EVALUATE AUTOSCALING SOLUTIONS IN SIMULATED EXPERIMENTS [98]

Name	Type	Description
SN1	Sine	$req(t) = 500 * sin(t\frac{\pi}{100}) + 700$
SN2	Sine	$req(t) = 1000 * sin(t\frac{\pi}{50}) + 1000$
ST1	Stair	$req(t) = 1000 * (1 + floor(t/100))$
ST2	Stair	$req(t) = 5000$ if $50 \leq t < 800$ else $req(t) = 50$
RP1	Ramp	$req(t) = 10t$ if $t < 800$ else $req(t) = 8000$
RP2	Ramp	$req(t) = 20t$ if $t < 800$ else $req(t) = 16000$
TW	Twitter	Real trace collected on January 1st, 2021 [50]

In order to determine and contrast the scalabilities, they have defined the Service Level Agreement (SLA) as being in the order of 0.6 seconds, meaning that any requests not processed within this SLA time frame amount to potential SLA violation; the application used for performance analysis of the scalability is AppDet, whereas all metrics have been taken against that application.

TABLE 4.2- AUTOSCALING APPROACHES FOR A SIMULATED SYSTEM ACROSS VARIOUS WORKLOADS TO MANAGE APPDET'S RESPONSE TIME [98]

		RT					
W	Approach	μ	σ	m	M	V	A_μ
SN1	*Static (1)*	0.98	0.27	0.50	1.31	846	1.00
	Rule-based	0.38	0.16	0.12	0.75	91	4.93
	Rule-based (+3)	0.41	0.15	0.16	0.88	110	4.99
	Step	0.28	0.12	0.10	0.45	0	7.00
	Target	0.43	0.15	0.17	0.79	145	4.29
	TargetFast	0.41	0.06	0.27	0.53	0	4.57
	ScaleX	**0.48**	**0.02**	**0.28**	**0.51**	**0**	**3.77**
	QN-CTRL	0.48	0.03	0.01	0.54	0	4.47
SN2	*Static (1)*	1.10	0.39	0.32	1.52	820	1.00
	Rule-based	0.37	0.21	0.05	0.70	181	7.68
	Rule-based (+3)	0.43	0.27	0.05	1.19	272	6.96
	Step	0.32	0.18	0.05	0.57	0	9.24
	Target	0.56	0.34	0.06	1.37	436	5.20
	TargetFast	0.43	0.10	0.18	0.67	37	8.14
	ScaleX	**0.46**	**0.06**	**0.29**	**0.55**	**0**	**5.90**
	QN-CTRL	0.48	0.04	0.01	0.64	2	7.14
ST1	*Static (1)*	1.65	0.17	1.19	1.82	1000	1.00
	Rule-based	0.58	0.10	0.29	0.67	624	20.68
	Rule-based (+3)	0.51	0.06	0.29	0.67	41	25.24
	Step	0.51	0.07	0.29	0.66	34	25.60
	Target	0.49	0.06	0.29	0.73	28	28.13
	TargetFast	0.53	0.25	0.11	1.00	369	147.57
	ScaleX	**0.49**	**0.03**	**0.30**	**0.59**	**0**	**27.97**
	QN-CTRL	0.48	0.03	0.01	0.77	2	29.68
ST2	*Static (1)*	1.32	0.66	0.16	1.74	755	1.00
	Rule-based	0.65	0.43	0.01	1.72	553	14.28
	Rule-based (+3)	0.47	0.31	0.01	1.69	188	18.94
	Step	0.54	0.41	0.01	1.69	228	18.35
	Target	0.46	0.27	0.01	1.71	101	18.88
	TargetFast	0.45	0.28	0.01	1.00	353	89.81
	ScaleX	0.41	0.14	0.07	0.87	9	19.98
	QN-CTRL	**0.44**	**0.11**	**0.01**	**1.72**	**5**	**19.73**
RP1	*Static (1)*	1.57	0.32	0.01	1.80	969	1.00
	Rule-based	0.62	0.07	0.01	0.67	842	17.14
	Rule-based (+3)	0.52	0.07	0.01	0.64	17	22.29
	Step	0.54	0.06	0.01	0.66	98	22.02
	Target	0.49	0.06	0.01	0.65	18	24.93
	TargetFast	0.52	0.26	0.01	1.02	381	134.38
	ScaleX	**0.48**	**0.05**	**0.01**	**0.50**	**0**	**24.86**
	QN-CTRL	0.52	0.15	0.01	1.39	62	25.19
RP2	*Static (1)*	1.70	0.27	0.01	1.87	982	1.00
	Rule-based	0.93	0.10	0.01	1.00	982	17.14
	Rule-based (+3)	0.54	0.06	0.01	0.90	24	43.44
	Step	0.60	0.14	0.01	0.97	235	43.23
	Target	0.53	0.10	0.01	0.95	112	48.70
	TargetFast	0.55	0.29	0.01	1.04	434	329.58
	ScaleX	**0.49**	**0.04**	**0.01**	**0.52**	**0**	**48.48**
	QN-CTRL	0.48	0.03	0.01	0.65	3	50.99
TW	*Static (1)*	1.71	0.03	1.66	1.79	1000	1.00
	Rule-based	0.30	0.03	0.25	0.39	0	52.32
	Rule-based (+3)	0.33	0.04	0.26	0.43	0	47.14
	Step	0.46	0.08	0.27	0.57	0	31.74
	Target	0.46	0.04	0.26	0.54	0	30.71
	TargetFast	0.54	0.26	0.12	0.99	436	131.34
	ScaleX	**0.47**	**0.04**	**0.29**	**0.52**	**0**	**30.11**
	QN-CTRL	0.48	0.03	0.01	0.57	0	35.56

Abbreviations of the columns of Table 4.2 are as follows:

W – Workloads
V – SLA violations
Aμ - Number of allocated cores
RT – Response time
μ - Average RT
σ – Standard deviation
m – Minimum value
M – Maximum value

In another study conducted [109], this work presents the outcomes of evaluating autoscaling performance for two-layer virtualization (VMs and pods) in the public clouds of AWS, Microsoft, and Google, using the specified approach and the Autoscaling Performance Measurement Tool. (APMT – Autoscaling performance measuring tool) developed by the authors. For the testing, four workload patterns were used: linear increase, linear increase with a constant, random, and triangular. Each test ran for 20 minutes, with a request timeout set to 6.5 seconds. A total of 50 simulated concurrent clients were used in each test.

The comparison results indicated that the performance characteristics of multilayered cloud applications are significantly influenced by factors such as the time taken to make scaling decisions, the underlying hardware of VM instances, and the level of synchronization between autoscaling across different virtualization layers. In this context, the GCE/Kubernetes solution demonstrated the best overall performance, which can be attributed to the overprovisioning of VMs.

TABLE 4.3 - PERFORMANCE COMPARISON BASED ON THE AMOUNT OF QOS VIOLATIONS [109]

Load Pattern	**Amount of latency breaks**			**Amount of errors breaks**		
	AWS	Azure	GCE	AWS	Azure	GCE
Linear Increase	0	0	0	**0**	**0**	382
Linear and Constant	17962	40934	**250**	25707	42725	**1251**
Random	1545	2570	**1127**	**1720**	2570	1835
Triangle	6418	15222	**76**	9954	16368	**845**

TABLE 4.4 - PERFORMANCE FOR AWS, AZURE, AND GCP AUTOSCALING SOLUTIONS FOR SCALE-OUT EVENTS [109]

Load Pattern	**Installation**	**Scale-out**	**Atoscaling Time, *seconds***
Linear Increase	AWS	1^{st}	28.06
		2^{nd}	9.03
	Azure	1^{st}	**128.00**
		2^{nd}	126.00
	GCE	1^{st}	8.01
		2^{nd}	12.01
Linear Increase and Constant	AWS	1^{st}	17.03
		2^{nd}	28.08
	Azure	1^{st}	**128.00**
		2^{nd}	123.00
	GCE	1^{st}	26.02
		2^{nd}	11.95
Random	AWS	1^{st}	33.08
		2^{nd}	15.01
	Azure	1^{st}	**131.00**
		2^{nd}	117.00
	GCE	1^{st}	11.01
		2^{nd}	7.99
Triangle	AWS	1^{st}	6.01
		2^{nd}	18.02
	Azure	1^{st}	**155.00**
		2^{nd}	128.00
	GCE	1^{st}	7.98
		2^{nd}	8.01

4.2 OpenStack-based autoscaling

Recent studies have shown various methodologies in the automation of scaling using Heat [78-84]. For a more detailed discussion on auto-scaling with Heat, see [78,79]; the policies regulating the process are detailed in [80]. The stack update process is explained in [81], laying the groundwork for developing advanced automated scaling systems for web servers. The document in [82] outlines the resource types in Senlin for Heat, which facilitate the creation of a comprehensive auto-scaling solution. It also includes a tutorial for users on how to design a Senlin cluster using Heat. Additionally, a detailed comparison of scaling on OpenStack through Heat and various orchestration tools is presented in [83]. According to Kaur et al. 2021, a comprehensive experiment was conducted on OpenStack networking and auto-scaling primarily using a Heat orchestration template [84]. They employed OpenStack Ussuri, released in 2020, and demonstrated a fully functioning neutron networking setup and autoscaling. They further showcased how the Heat orchestration template is used to define the networking infrastructure and optimize scalability. The following material describes the deployment of autoscaling using Heat and Ceilometer [85–88]. Gupta et al. discussed how OpenStack utilizes Heat and Ceilometer to facilitate auto-scaling in [85]; Yang et al. proposed a new methodology for autoscaling with Heat and Ceilometer in [86]; and Gomez-Rodriguez et al. exchanged Ceilometer for Monasca, which performed better, yet creates integration headaches [89]. A comprehensive explanation of this autoscaling approach is in [90]. In 2021, Lanciano et al. proposed predictive autoscaling with Monasca, along with an architecture for autoscaling cloud services based on the prediction of the system's future state [91]. In [93], Rahman et al. examine how factors like the start-up time of underlying virtualization technologies influence both the Quality of Service (QoS) and cost savings. In [95, 96], deep reinforcement learning and federated learning are applied to auto-scaling, respectively. In a later study presented in [93], Proposed an auto-scaling mechanism that utilizes machine learning to predict the resource demand of a cloud application. Their approach uses a neural network to forecast resource usage and adjust the number of VMs accordingly. The proposed system outperforms other existing autoscaling techniques in terms of resource utilization and response time.

4.2.1 Auto-Scaling Using Heat

Heat module of OpenStack has been used to supervise and manage the cloud infrastructure and services [78]. Heat orchestration service in OpenStack is primarily to give a unified, easy interface for end users and automated systems to manage their life cycle of applications and infrastructure in the Cloud. The heat templates serve as a guide for defining and provisioning cloud resources. Among its several features, such as the support for autoscaling, Heat integrates with cloud’s telemetry service to realize autoscaling. In 2012, OpenStack developers introduced a practical implementation of autoscaling using Heat, which operates by setting and observing alarms to scale compute instances dynamically [79]. In version 1.00, Heat adopted a straightforward

method for implementing autoscaling (refer to Figure 4.1). For each instance it provisions, Heat also generates a lightweight Python script, commonly referred to as push-stat. The script is configured to execute at regular intervals using a Cron job. This process essentially uses the 'Util' package to collect the basic performance metrics like CPU and memory usage from some active instances and delivers that data to CW-lite. In turn, Heat regularly checks on the parameters reported by those instances. When either of the instances crosses a defined threshold, Heat triggers a scale-out or scale-in based on the defined autoscaling policy (see Figure 4.1).

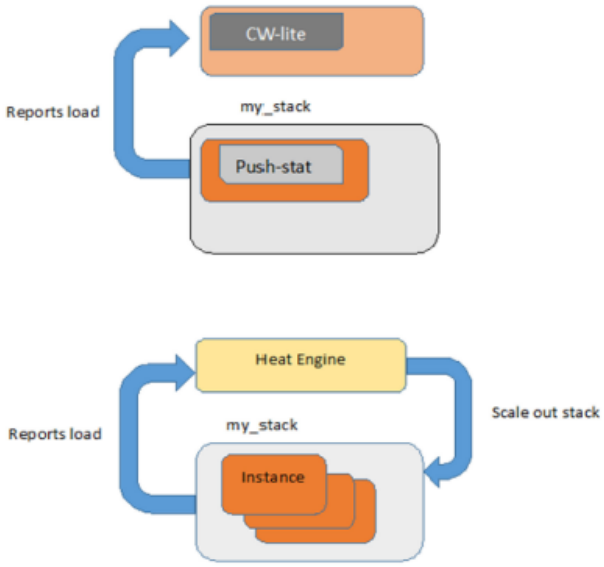

Fig. 4.1: Auto-scaling in Heat

There are many difficulties with using Heat alone for OpenStack autoscaling.

- Inflexible Metrics
- Lack of Historical Data
- Slow Response Times
- Lack of resource optimization

4.2.2 Auto-Scaling Using Heat and Ceilometer

An alternative method for implementing auto-scaling has been proposed by Yang et al. in 2016 [86]. Instead of monitoring data storage and the creation of an alarm directly, this model suggests using Ceilometer to carry out all these functions. Heat instructs Ceilometer to create and evaluate alarms, and on this account, Ceilometer only collects information relevant to Heat, thus minimizing the data collection process.

Alarms are created and defined by Heat templates that contain certain rules such as threshold limits, evaluation time windows, and comparison operators. Thus, the alarms are managed with the help of the Ceilometer REST API so that the administrator can monitor and operate on the alarm lifecycle. The process of creating, evaluating, and notifying alarms is shown in Figure 4.2. The Ceilometer Evaluator invokes the Ceilometer API and retrieves the alarm definition rules as they were originally defined

in the Heat template stored in the Ceilometer database. It takes stock of whether or not the already specified thresholds have been breached. On activating a threshold, the evaluator uses the RPC Notifier API so that an alarm can be raised. The Ceilometer Notifier, keeping an eye on the RPC bus, performs the respective actions. In this case, the outside system, such as Heat, can be informed by the Ceilometer's Notifier to trigger an auto-scaling action by means of a configured Uniform Resource Locator (URL) [86-88]. To begin with, from the onset, Ceilometer had a primary objective of collecting and storing time series data, but at the time of its initial development, there were no complete specifications on systems for data manipulation, querying, and management. Consequently, the adoption of a very flexible data model was the result of this lack of specification. Although this flexibility has provided an adaptive and useful feature for the Ceilometer, it has also caused performance drawbacks, particularly for sustained data retention. Hence, storing huge amounts of data for long periods created difficulties for back-end databases. Some of the main drawbacks of Ceilometer include:

1. High storage requirements
2. Inefficiencies in data ingestion
3. Suboptimal performance of the query API

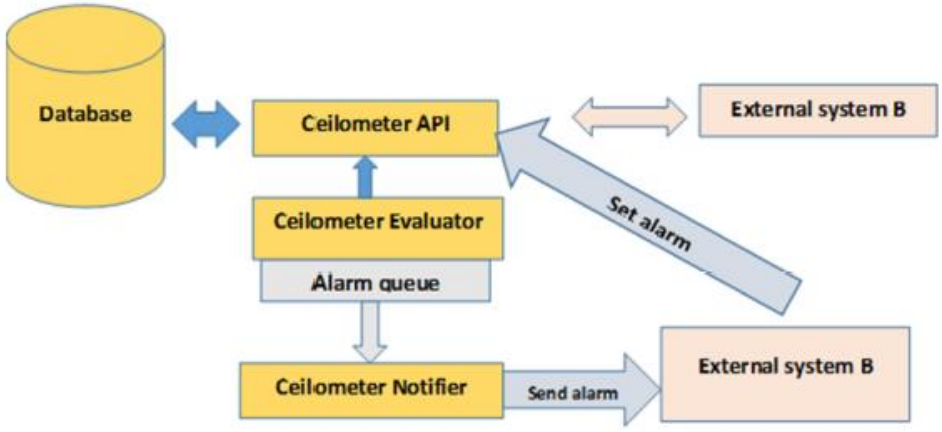

Fig. 4.2: Alarm generation and assessment in Ceilometer

Heavily relies on Ceilometer to generate alarms, which is essential to enable instance autoscaling. But caps with the limitations of Ceilometer bring many complexities with it, especially while deploying the solution. For instance, it can really take time when the Ceilometer evaluator requests alarm definitions or data from the Ceilometer API on the templates provided by Heat because the database contains a large amount of data. Indefinitely this will lead to long delays in response times and possibly time-outs at the API, resulting in a failure of operations. In real cases, if the Ceilometer is used as a data collector, timely data response will not come from it to facilitate any effective handling of critical events by the administrators, thus defeating the purpose of using the Ceilometer.

4.2.3 Auto-Scaling Using Heat and Monasca

In 2017, a series of comparative tests on Monasca and Ceilometer were executed by Gomez Rodriguez et al. [89]. Monasca, for monitoring as a service, is designed and implemented as a highly available and horizontally scalable service; hence, it is considered high performance. Apache Kafka is the core component of Monasca, being used for message queuing, while Infra DB is the time series database. It is multi-tenant, meaning it can store measurements, alarm definitions, and notifications. Monasca integrates with Heat to perform autoscaling. The authors evaluate the performance of Ceilometer and Monasca based on the time spent retrieving monitoring data and on comparisons of the memory used by both systems. Monasca has proven to perform better than Ceilometer in terms of both memory consumption and time spent retrieving monitoring data. The downside of Monasca being less coupled to OpenStack than Ceilometer is that it is considerably harder to integrate with. Documentation on integrating Monasca with other components is lacking. Deploying an OpenStack cloud is a challenge as it integrates various components working together in harmony to create a fully operational cloud. Monasca must improve either the integration aspects or the documentation to be considered a substitute or an alternative to use along with Ceilometer.

4.2.4 Auto-Scaling Using Senlin

In the online article [110], they have shown how to work with OpenStack Senlin clustering. Ansible was used as the deployment language to deploy the clustering service. In this article, the authors have discussed many core functionalities of Senlin for example, verifying Senlin installation status, creating clusters using server profiles, exercising cluster auto-healing, and cluster expanding and shrinking practices are also performed. Another article [111] written by the same author, describes how to approach Senlin autoscaling. It contains a complete guide on how to perform autoscaling using webhooks.

In a Medium article [112], the author discusses how to reach Auto-scaling OpenStack Instances with Senlin and Prometheus. The article specifically outlines how to configure a Senlin cluster that scales out when Prometheus determines that the idle CPU has fallen below a certain threshold. The main tasks that were carried out in this article are as follows,

- Construct a Senlin cluster using a single virtual machine and a webhook so that the cluster may grow.
- Install Prometheus in a virtual machine (VM).
- Configure Prometheus to scrape Senlin cluster nodes.
- Configure Prometheus to notify us if our Senlin cluster's total idle CPU falls below 50%.
- Configure Alertmanager to scale out when it gets a Prometheus alarm by calling the Senlin webhook.

- Make sure the Senlin cluster is scaled out by creating a fictitious load in our Senlin node.

4.3 Non-OpenStack-based autoscaling

An approach to auto-scaling in cloud environments is presented in [26]. In their paper ACAS: An Autonomic Cloud Application Scaling System. The authors proposed a hierarchical architecture that integrates application-level and system-level auto-scaling policies to optimize resource utilization and reduce energy consumption. For this work, the CloudSim Framework has been used.

Dutreilh et al. [27] analyze both threshold approaches and reinforcement learning in horizontal autoscaling. In [28], a technique is reported that helps scale resources at each particular level along with VM scaling with the aim of providing better resource efficiency and lower costs for cloud providers. The additional thresholds with three critical metrics, such as CPU load, response time, and network bandwidth, are appended to the standard two-threshold model by Hasan et al. [29] for scaling decisions. A dynamic scaling strategy for PaaS and SAAS applications is explained by Chieu et al. [30] where the scalability on numbers of VMs is adjusted according to the active session count. Scaling occurs when all instances have crossed predefined limits. It is quite simple yet highly reliant on the definition of the thresholds for efficacy.

Tesauro et al. [31] developed a hybrid learning system, designed for dynamic server allocation, seeking profit maximization. Their approach integrates a queuing network model for real-time management with reinforcement learning, specifically, the SARSA algorithm for offline training. In this system, allocation decisions are made on the basis of workload demands and application response times. Likewise, Rao et al. [32] propose VCONF, another reinforcement learning-based approach to implement neural networks. It automatically tunes VM's configuration parameters. VCONF automatically adjusts cloud VM settings to application requirements, delivering maximum performance for hosted services.

Ali-Eldin et al. [33] employed the queuing theory for developing a model of a cloud service using dual adaptive hybrid controllers that mix reactive and proactive strategies to retain the prediction of future workloads and supplement elasticity. Padala et al [34]. develop a feedback-based resource management system that automatically adjusts itself with changing workloads to serve its service level objectives. This approach includes live model estimation, through which the mapping between applications and resources is continuously updated, along with a two-tier multi-input, multi-output (MIMO) controller that dynamically allocates resources to applications.

Bodik et al. [35] presented a method based on statistical machine learning to predict system performance in a single-tier application. Liu et al. [36] examined a cloud application scaling mechanism, which manages CPU and bandwidth utilization according to instantaneous workload peaks to upgrade VM type dynamically. Most of them have auto-scaling implemented recently by Michael et al. [37]: bioinformatics

and biomedical applications are now being run on the cloud with accelerated execution times. Victor et al. [38] have developed a regression based performance model that predicts the throughput of NoSQL databases for resource scaling to meet SLA requirements. Wajahat et al. [39] integrated neural networks for learning performance behavior models and linear regression to estimate the impact of scaling decisions into a machine learning-based auto-scaling approach. An adaptive fuzzy logic controller is implemented by Persico et al. [40] based on the usage of CPU and bandwidth in horizontally scaling a cloud application.

In their comparative analysis of auto-scaling techniques for cloud applications with complex workflows, Alexey et al. [41] discovered quite a few; Salah et al. [42] provided an analytical approach that uses Markov chains and queueing theory to horizontally scale firewall services so that performance SLOs are guaranteed; and Tania et al. [43] conducted an extended survey of auto-scaling techniques for cloud applications. Another review study [44] examined the management of cloud resources and identified some fundamental challenges, one of which is to achieve consistent and predictable performance for cloud-deployed applications.

More and more researchers have recently begun to look at automated resource provision for multi-tier applications. Song et al. [45] proposed a hybrid auto-scaling strategy for cloud-hosted multi-tier applications. The method involves horizontal scaling for long-term workload management according to historical traffic patterns, while vertical scaling seeks to reduce SLO violations during sudden workload bursts. Chenhao et al. [46] presented a fault-tolerance model to minimize operational costs for web applications that utilize AWS EC2 spot instances. They implemented the mixed usage of fault tolerance with auto-scaling policies to save costs considerably on the AWS Cloud. Adnan et al. [97] investigated a variety of ways of provisioning VMs to support multiple applications on a shared infrastructure. These methods-centered around three approaches-consist of pure reactive provisioning, a hybrid of reactive and proactive, and session-based adaptive admissions control.

CHAPTER 5

METHODOLOGY

In this chapter, all the steps used to deploy the OpenStack cloud, implement the auto-scaling solutions, install and configure the monitoring setup, and all the related configurations are discussed in detail.

5.1 Install OpenStack private cloud

For the installation of OpenStack, the DevStack all-in-one method is used. DevStack consists of highly dynamic scripts that can be used for installing a complete OpenStack environment directly from the Git repositories. Moreover, it acts as the interactive development setup and the base for most of the functional testing done across the OpenStack project. In default settings, only the basic OpenStack projects are installed, which include Keystone, Glance, Nova, Placement, Cinder, Neutron, and Horizon. But in our case, it was required to have more advanced projects other than the default basic projects, which include Heat (Orchestration), Ceilometer (Telemetry), Gnocchi (Time series database), Aodh (Alarming), and Senlin (Clustering).

The implementation of OpenStack is done on a bare-metal PC. The specification of that PC is as follows.

- Operating System – Ubuntu Server 22.04 (Jammy)
- OpenStack version – Zed
- Memory (RAM) – 28GB
- Hard Disk Capacity – 320GB
- Processor – Intel Core i3
- Network Type – Static IP

Ubuntu Server 22.04 operating system is installed as the host operating system in the PC.

Fig 5.1 Host OS information

In the Host OS, it is required to create a user-named stack, and that user needs the sudo privileges to deploy OpenStack using the DevStack Method.

- Create the stack user and set its home directory.
  useradd -s /bin/bash -d /opt/stack -m stack

- Provide required permissions for the home directory.
  chmod +x /opt/stack

- Giving the user sudo privileges and switching to the stack user.
  echo stack ALL=(ALL) NOPASSWD: ALL | tee /etc/sudoers.d/stack
  su -u stack -i

- Get DevStack and select the main branch.
  git clone https://opendev.org/openstack/devstack
  cd devstack
  git checkout -b unmaintained/zed origin/unmaintained/zed

- Create the local.conf file – This is the configuration file which used to install and customize different OpenStack services based on the requirements. When DevStack is running it reads this file to determine various settings and options for the installation process. The content of the local.conf file which was used to deploy OpenStack in this project is below.

```
stack@dinith:~/devstack$ cat local.conf
[[local|localrc]]
ADMIN_PASSWORD=dinith
DATABASE_PASSWORD=$ADMIN_PASSWORD
RABBIT_PASSWORD=$ADMIN_PASSWORD
SERVICE_PASSWORD=$ADMIN_PASSWORD

enable_service h-eng h-api h-api-cfn h-api-cw
enable_plugin heat https://opendev.org/openstack/heat unmaintained/zed
enable_plugin heat-dashboard https://opendev.org/openstack/heat-dashboard unmaintained/zed
enable_plugin ceilometer https://opendev.org/openstack/ceilometer unmaintained/zed
CEILOMETER_BACKEND=gnocchi
CEILOMETER_NOTIFICATION_TOPICS=notifications,profiler
enable_plugin aodh https://opendev.org/openstack/aodh unmaintained/zed
enable_plugin senlin https://opendev.org/openstack/senlin unmaintained/zed
enable_plugin senlin-dashboard https://opendev.org/openstack/senlin-dashboard unmaintained/zed
enable_plugin octavia https://opendev.org/openstack/octavia unmaintained/zed
HOST_IP=192.168.204.130
stack@dinith:~/devstack$
```

Fig 5.2 local.conf file content

After mentioning the required services and plugins in the local.conf file, the installation of the OpenStack cloud was started by executing the ./stack.sh command, which is a Python script that installs many git trees and packages during this process. As per the documentation, the estimated completion time was around 15-30 minutes for default services. But in this case, we mentioned extra services and plugins other than the default services, and based on the Internet speed, it took nearly 1.5 hours to complete the OpenStack cloud installation process.

```
This is your host IP address: 192.168.204.130
This is your host IPv6 address: ::1
Horizon is now available at http://192.168.204.130/dashboard
Keystone is serving at http://192.168.204.130/identity/
The default users are: admin and demo
The password: dinith

Services are running under systemd unit files.
For more information see:
https://docs.openstack.org/devstack/latest/systemd.html

DevStack Version: zed
Change: 0f18f54cef0c193d7c082a0e6c6884346f1f09a7 Replacing usage of rdo-release rpm 2024-08-06 16:05:47 +0530
OS Version: Ubuntu 22.04 jammy

2024-09-10 19:24:24.464 | stack.sh completed in 4469 seconds.
stack@dinith:~/devstack$
```

Fig 5.3 OpenStack installation completed

The installation of the services and plugins can be verified by using the OpenStack service list command. Below is the output of our setup.

```
stack@dinith:~$ openstack service list
+----------------------------------+-------------+----------------+
| ID                               | Name        | Type           |
+----------------------------------+-------------+----------------+
| 05f7b24f955f44e3b73e507aa5c7834b | cinder      | block-storage  |
| 09e5c93c2df940bfa6dab11a97cb7103 | neutron     | network        |
| 23fc678c18984682b8ccd7fa7298358a | gnocchi     | metric         |
| 5ea9d848ee33411ebf2899ff1ab89b06 | cinderv3    | volumev3       |
| 8052478d912e4cb584f21e1168f31044 | senlin      | clustering     |
| 891c612d2f2843b984fc727e491c243b | glance      | image          |
| a36416f0cca1408292634754e3ac596d | nova        | compute        |
| b77e84de150449a48559b6c4a0084212 | nova_legacy | compute_legacy |
| e7fc5bc58e484f659b15f97c6552d862 | heat-cfn    | cloudformation |
| edbc6f00c86245248deecd14cfc61988 | keystone    | identity       |
| f8a04f5775574d94b36a562c1e45ad0b | placement   | placement      |
| fa92a02e25ad4799a0fd88dbc6fc80d1 | heat        | orchestration  |
| fec396a2d16e4b8ea1bbbce4985bb294 | aodh        | alarming       |
+----------------------------------+-------------+----------------+
```

Fig 5.4 OpenStack services verification

The Horizon dashboard (GUI) can be accessed using the host address http://192.168.204.130/dashboard/.

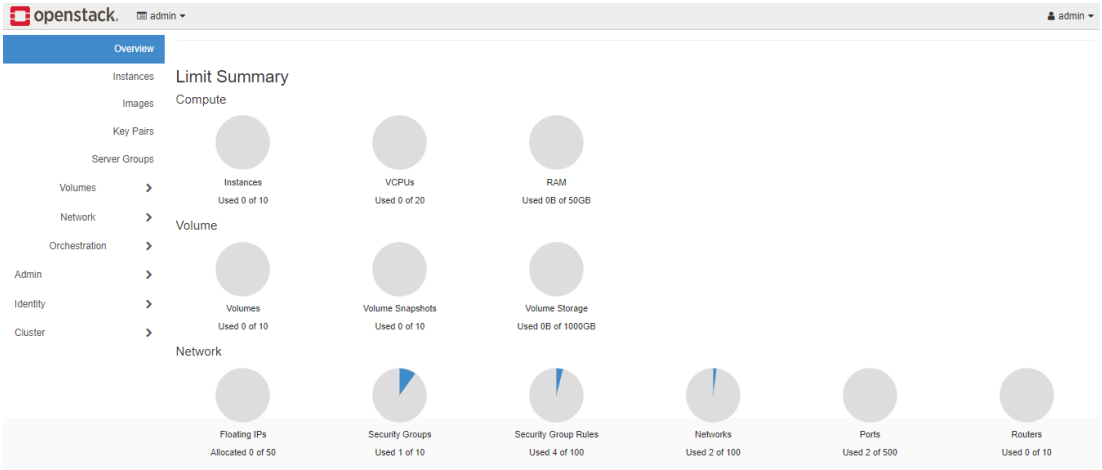

Fig 5.5 OpenStack Horizon dashboard

5.2 Install and configure the monitoring setup

In this work, setting up monitoring is a vital part because to measure the performance of the autoscaling solutions, we need to have a proper monitoring setup. The number of VMs created, the individual matrices of those VMs, and the time that VMs are spawned/dead are some of the major factors that need to be monitored. For reporting purposes and performance comparison of autoscaling solutions, this monitoring setup is a must. In the monitoring setup of this project, Grafana, Prometheus, and OpenStack-exporter were used.

5.2.1 Installing Prometheus

In this section, the procedure of installing Prometheus and the configuration of Prometheus are discussed in detail.

- Install Prometheus using the ‘apt install’ command

```
stack@dinith:~$ sudo apt install prometheus
```

Fig 5.6 Installing Prometheus

- The installation of the Prometheus service can be verified using the ‘systemctl status prometheus’ command

```
stack@dinith:~$ systemctl status prometheus
● prometheus.service - Monitoring system and time series database
     Loaded: loaded (/lib/systemd/system/prometheus.service; enabled; vendor preset: enabled)
     Active: active (running) since Sat 2024-09-21 03:57:19 UTC; 11min ago
       Docs: https://prometheus.io/docs/introduction/overview/
             man:prometheus(1)
   Main PID: 15009 (prometheus)
      Tasks: 10 (limit: 15378)
     Memory: 36.2M
        CPU: 3.957s
     CGroup: /system.slice/prometheus.service
             └─15009 /usr/bin/prometheus
```

Fig 5.7 Verification of Prometheus Installation

- The Prometheus service uses the default port of 9090. Also, in the default setup, Prometheus is exporting the metrics of the host machine. For that purpose, Prometheus is using port 9100.

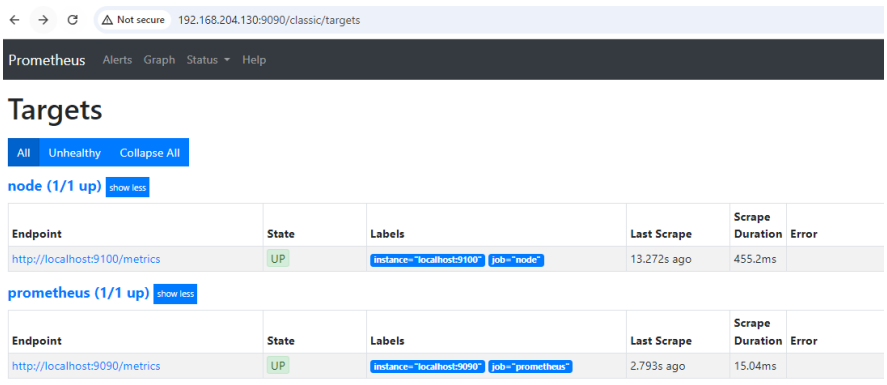

Fig 5.8 Prometheus targets and ports

- All the scraped metrics from the host machine by Prometheus can be viewed using the curl http://192.168.204.130:9100/metrics command.

```
stack@dinith:~$ curl http://192.168.204.130:9100/metrics
# HELP apt_autoremove_pending Apt package pending autoremove.
# TYPE apt_autoremove_pending gauge
apt_autoremove_pending 0
# HELP apt_upgrades_pending Apt package pending updates by origin.
# TYPE apt_upgrades_pending gauge
apt_upgrades_pending{arch="all",origin="Ubuntu:22.04/jammy"} 1
apt_upgrades_pending{arch="all",origin="Ubuntu:22.04/jammy-updates"} 23
apt_upgrades_pending{arch="amd64",origin="Ubuntu:22.04/jammy-updates"} 48
# HELP go_gc_duration_seconds A summary of the pause duration of garbage collection cycles.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 3.5707e-05
go_gc_duration_seconds{quantile="0.25"} 0.00011148
go_gc_duration_seconds{quantile="0.5"} 0.000275984
go_gc_duration_seconds{quantile="0.75"} 0.000734766
go_gc_duration_seconds{quantile="1"} 0.023389079
go_gc_duration_seconds_sum 0.371686777
go_gc_duration_seconds_count 505
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 8
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.18.1"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
```

Fig 5.9 Scraped metrics by Prometheus

5.2.2 Installing Grafana

Even though Prometheus is a great monitoring tool, it consists only of a basic data visualization facility, and to visualize data, there are a lot of manual settings that need to be done, like writing PromQL queries. As a solution for this, in this project, Grafana dashboards are used. The metrics scraped from Prometheus are pipelined to the Grafana endpoint. In Grafana, those metrics are visualized in a very fine and clear way. And there are a lot of customization facilities available on Grafana. As a result of that the graphs and dashboards can be customized based on our requirements. In this section, the procedure of installing Grafana and the configuration of settings are discussed in detail.

- The procedure for installing Grafana:
    - Package information is updated.
        - apt-get install -y apt-transport-https
        - apt-get install -y software-properties-common wget
        - wget -q -O - https://packages.grafana.com/gpg.key | apt-key add –
    - Stable repos are added related to Grafana
        - echo deb https://packages.grafana.com/enterprise/deb stable main | tee -a /etc/apt/sources.list.d/grafana.list
    - Repositories are updated, and Grafana is installed.
        - apt-get update
        - apt-get install grafana-enterprise

- Grafana Server is started.
  - systemctl daemon-reload
  - systemctl start grafana-server
  - systemctl status grafana-server

- Enable Grafana at boot time.
  - systemctl enable grafana-server.service

By completing the above steps, a fully functional Grafana server is implemented. By default, the Grafana server is served using port number 3000.

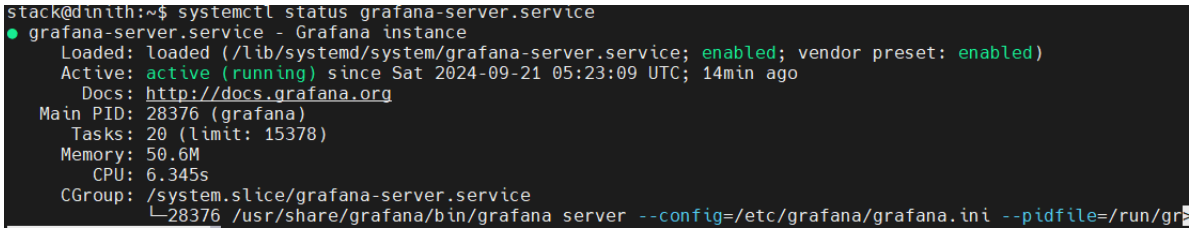

Fig 5.10 Verification of Grafana Server Installation

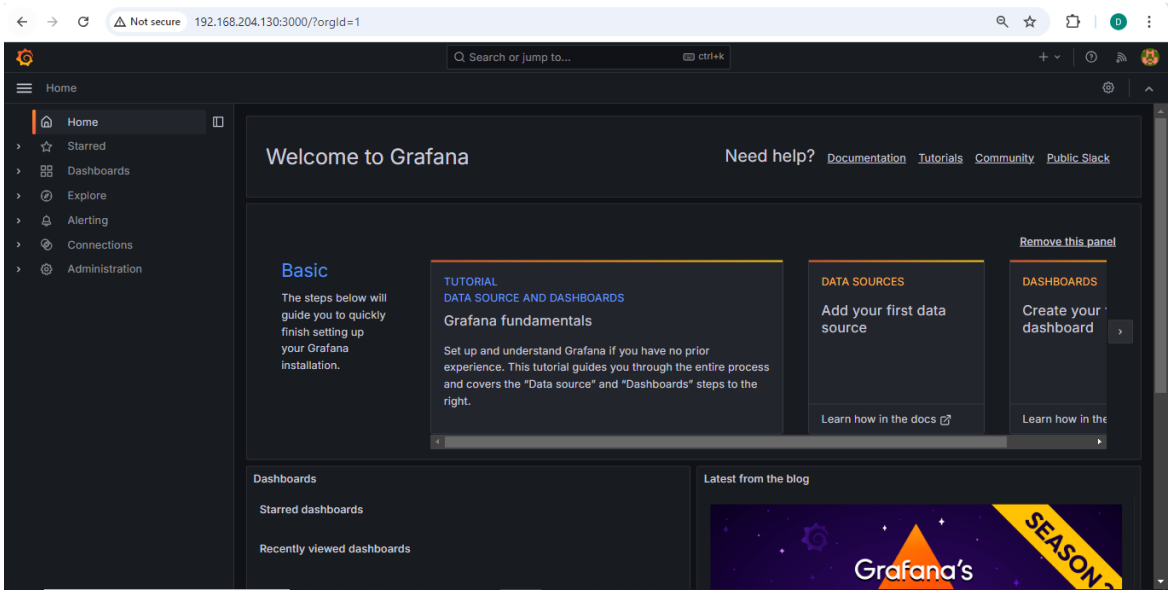

Fig 5.11 Grafana Dashboard

5.2.3 Installing OpenStack-exporter

OpenStack-exporter is used for monitoring the metrics of the OpenStack VMs. Using this tool, a lot of valuable information could be obtained. For example, the number of VMs, the Number of networks, the Number of Projects, etc. This is very important in our work as we need the exact behaviors of the OpenStack components when we are observing the performance of autoscaling solutions. In the installation process, we first deploy the OpenStack exporter using Docker images, then we convert that to a system

service and set it as a default service which will automatically start once the host PC is starting.

- The procedure of installation of OpenStack-exporter
    - Pulling the latest OpenStack-exporter from GitHub.
        - podman pull ghcr.io/openstack-exporter/openstack-exporter:latest

```
stack@dinith:~$ podman pull ghcr.io/openstack-exporter/openstack-exporter:latest
Trying to pull ghcr.io/openstack-exporter/openstack-exporter:latest...
Getting image source signatures
Copying blob 058cf3d8c2ba done
Copying blob e8d9a567199d done
Copying blob 33e068de2649 done
Copying blob b6824ed73363 done
Copying blob 7c12895b777b done
Copying blob a85ce7502e1a done
Copying blob 4aa0ea1413d3 done
Copying blob 27be814a09eb done
Copying blob da7816fa955e done
Copying blob 5664b15f108b done
Copying blob 9aee425378d2 done
Copying blob 903010886edf done
Copying blob 209150bce63e done
Copying blob 8fa1a6285903 done
Copying config 46d8865073 done
Writing manifest to image destination
Storing signatures
46d8865073f011ba5933b80add2d0959452f21d505531f6dcb8ef1ff522315cb
```

Fig 5.12 Pulling the latest OpenStack-exporter from GitHub

```
stack@dinith:~$ podman images
REPOSITORY                                  TAG         IMAGE ID      CREATED     SIZE
ghcr.io/openstack-exporter/openstack-exporter  latest      46d8865073f0  8 days ago  40.3 MB
```

Fig 5.13 Verification of Docker image download

- Creating the clouds.yaml configuration file – This file is mapped with the container once the container is running.

```
clouds:
  admin:
    region_name: RegionOne
    identity_api_version: 3
    # identity_interface: public
    auth:
      username: admin
      password: dinith
      project_name: admin
      project_domain_name: default
      user_domain_name: default
      auth_url: http://192.168.204.130/identity
~
```

Fig 5.14 The content of cloud.yaml file

- Start the OpenStack-exporter as a Container
    - podman run -v $HOME/clouds.yaml:/etc/openstack/clouds.yaml -it -p

9180:9180 ghcr.io/openstack-exporter/openstack-exporter:latest admin

The above command will start the OpenStack-exporter as a container and start to scrape the metrics related to OpenStack components. The available metrices can be viewed using the curl http://192.168.204.130:9180/metrics command on CLI or using http://192.168.204.130:9180/metrics on a web browser.

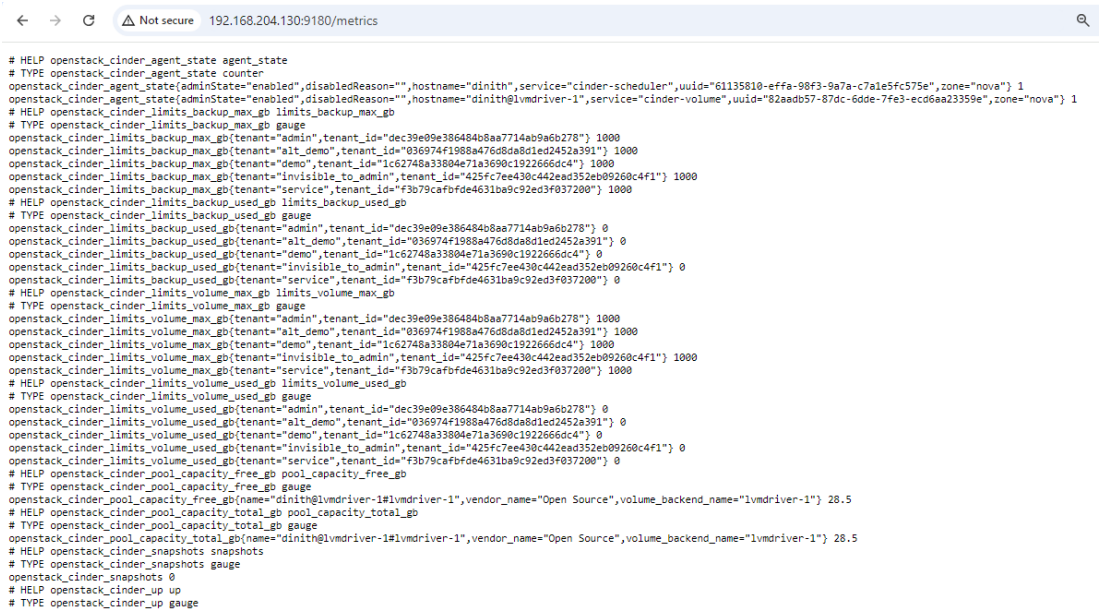

Fig 5.15 OpenStack-related metrics

- Convert the Docker container into a default system service and set it to automatically start when the host PC starts.
    - podman generate systemd e8333b08361d > /etc/systemd/system/uom-exporter.service
    - podman stop eager_hawking ##Stop the running container##
    - systemctl daemon-reload
    - systemctl start uom-exporter.service
    - systemctl enable uom-exporter.service

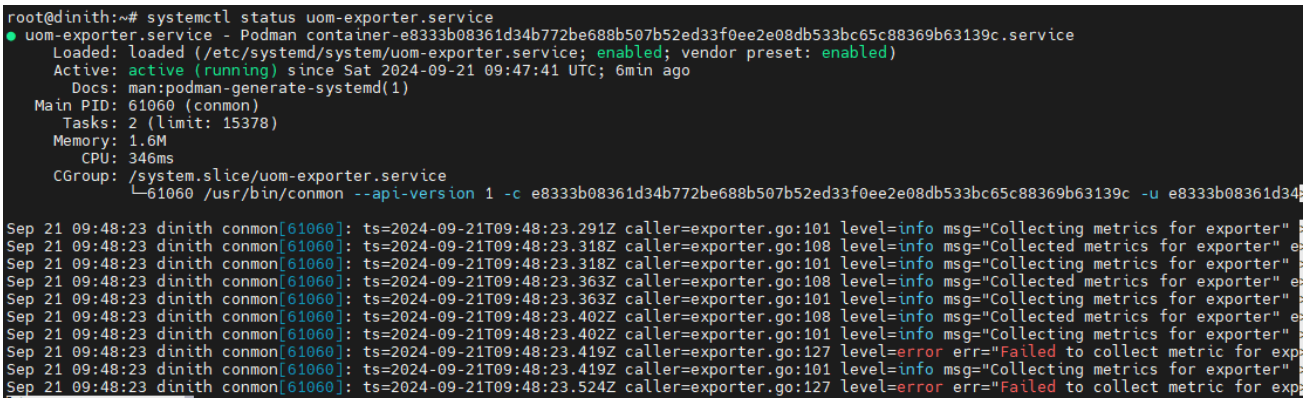

Fig 5.16 Status of the custom build system service using the container

5.2.4 Integration of Prometheus, Grafana, and OpenStack-exporter

In this section, the integration of the above services to obtain a completely functioning monitoring setup is discussed.

- Integrate node-exporter and OpenStack-exporter services with Prometheus.
    - The endpoints of the node-exporter (Port 9100) and the OpenStack-exporter (Port 9180) need to be configured in the /etc/prometheus/prometheus.yml file, then the Prometheus services should be restarted.

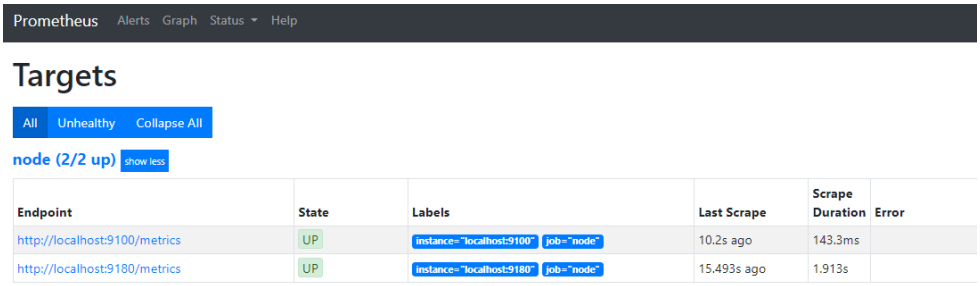

Fig 5.17 Exporters are integrated with Prometheus

- Visualize the exporter metrics scraped from Prometheus using Grafana
    - The data available at Prometheus needs to be integrated into Grafana as a data source.

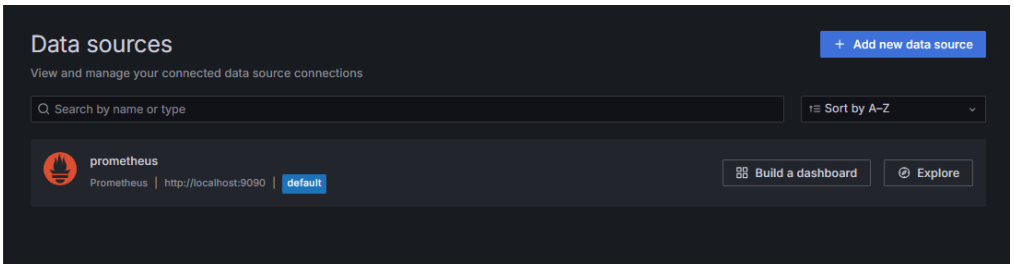

Fig 5.18 Prometheus data source is added in Grafana

    - Finally, dashboards are imported from the official Grafana website to visualize the exporter data.

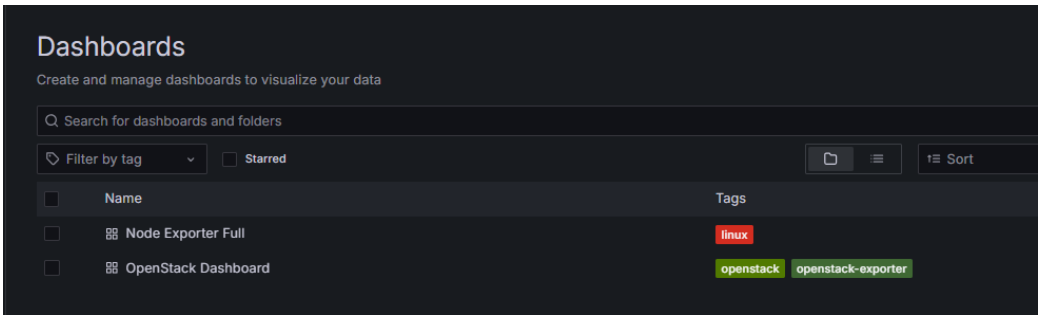

Fig 5.19 Dashboards are configured to visualize data related to relevant exporter data.

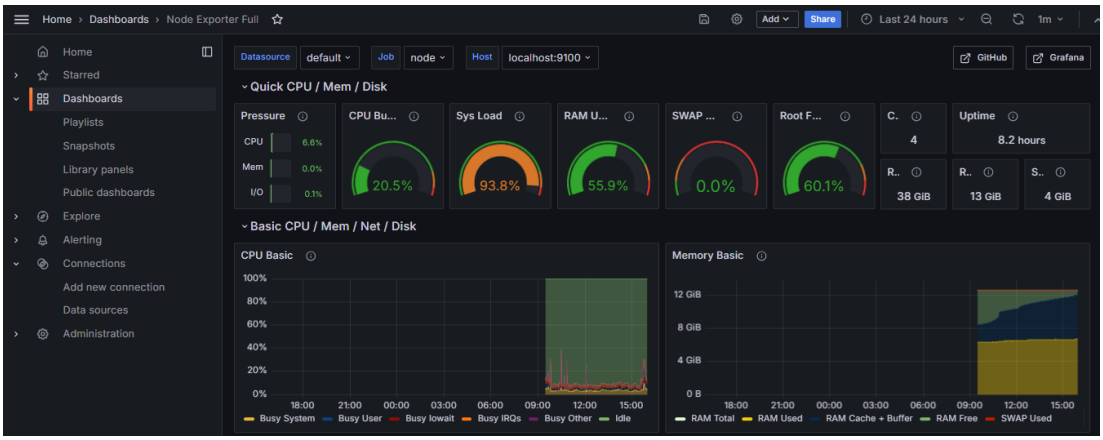

Fig 5.20 Node-exporter data visualization

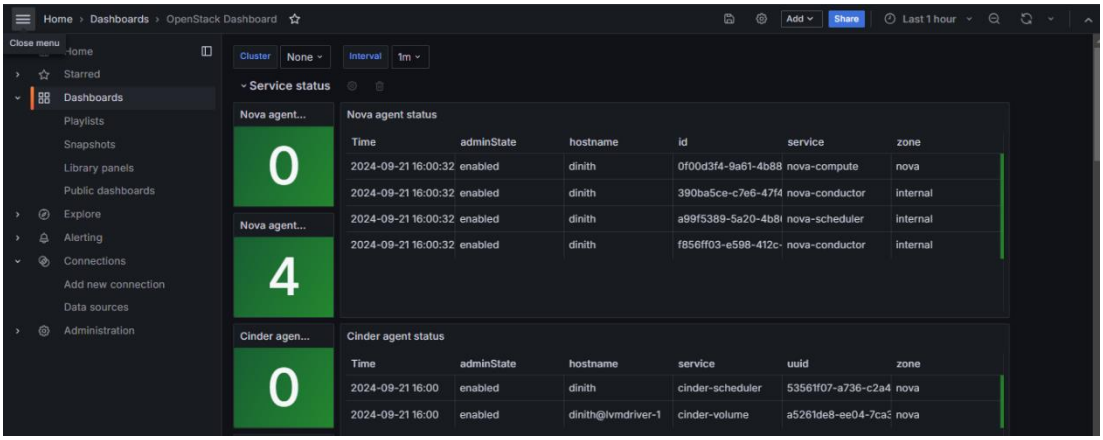

Fig 5.21 OpenStack-exporter data visualization

5.3 Deploy and prepare the VM

The main goal of this project is to compare the performance of autoscaling solutions, namely, autoscaling using Heat, Ceilometer, and Gnocchi and autoscaling using Senlin. To test the performance of these solutions, it is required to reference a VM that acts as a Webserver. In this section, the procedure of creating the Webserver VM, installing webserver packages, and integrating it into the monitoring setup is discussed.

For this purpose, an Ubuntu server image, which is optimized for cloud environments, is used. For this image, 1GB of block storage partition and 1GB of system RAM are allocated from OpenStack resources. For the creation of the VM, an instance flavor called m1.small is used, which consists of 2GB of RAM and 20GB of disk space. A private shared network is used for network connectivity, and a key pair is created to connect with the VM from the host network. A floating IP is associated with the VM to communicate with the host PC.

After creating the VM, all the required packages were installed. Below is the list of packages and their purposes.

- Web server – apache2 is installed as the web server using sudo apt-install apache2 command. Using the benchmark tool (Apache JMeter), this web server will be benchmarked.
- Prometheus exporter – This package is installed using sudo apt-install prometheus command. The purpose of this package is to monitor the resources like RAM and CPU when the webserver is benchmarked using the benchmarking tool.

After installing the required packages, this web server is also added to the monitoring setup to visually monitor the behavior. Now the setup is ready for the implementation of autoscaling solutions and testing.

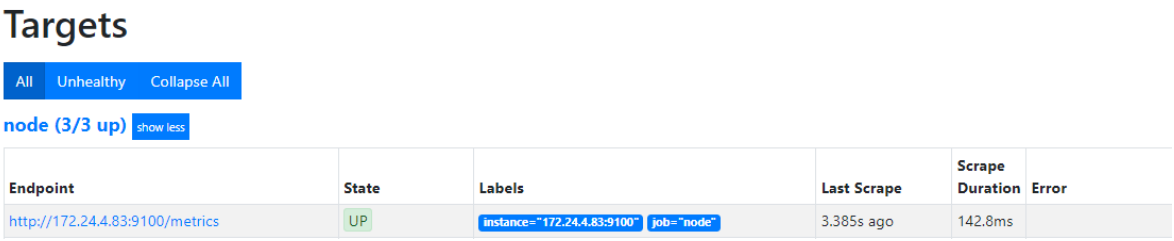

Fig 5.22 Web server integration to Prometheus

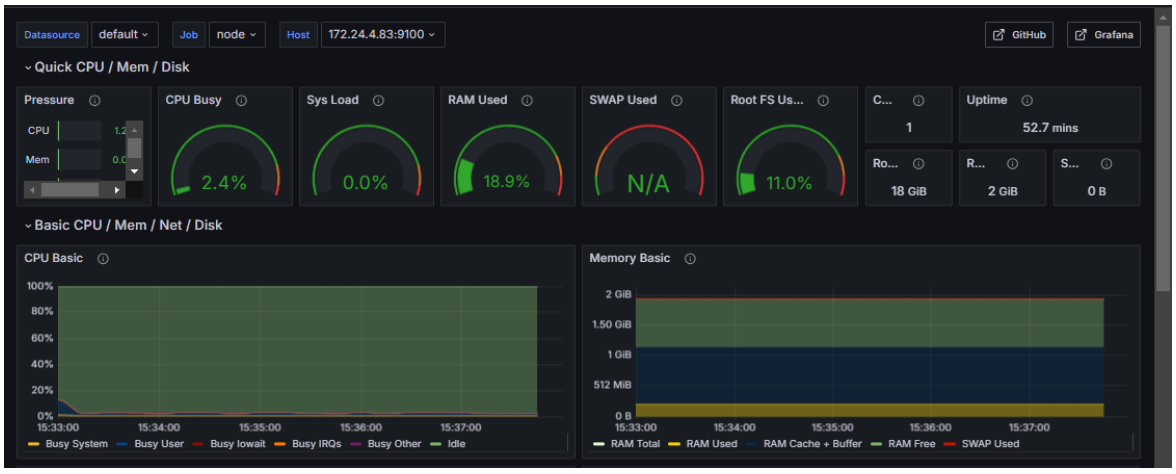

Fig 5.23 Web server metric visualization using Grafana

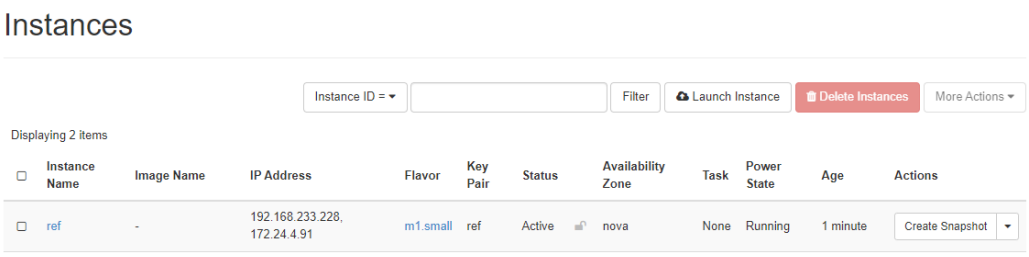

Fig 5.24 Web server instance on Horizon dashboard

5.4 Implementation of autoscaling solutions

5.4.1 Heat, ceilometer, and Gnocchi

In this approach, the autoscaling solution will be implemented using Ceilometer, Heat, and Gnocchi. Heat will be the autoscaling engine. The heat template will be written based on the Web server VM that we have created, and for writing the heat template, the YAML language is used. YAML is a user-friendly data serialization format commonly used for creating configuration files [113]. This Web server acts as a reference point where it is continuously monitored and based on the threshold, an alarm will be triggered for example, if the CPU utilization of the web server is higher than 80%, a high alarm will be triggered, this triggering will call its respective scaling policy and based on the scaling policy configuration, the auto-scaling group will be modified. On the other hand, if the CPU threshold is less than 20%, a low alarm will be triggered, similarly, this triggering will call its respective scaling policy, and based on the scaling policy configuration, the auto-scaling group will be modified as in the first scenario. The complete heat template is appended in Appendix A. The architecture diagram of this implementation is visualized in Figure 5.25 below.

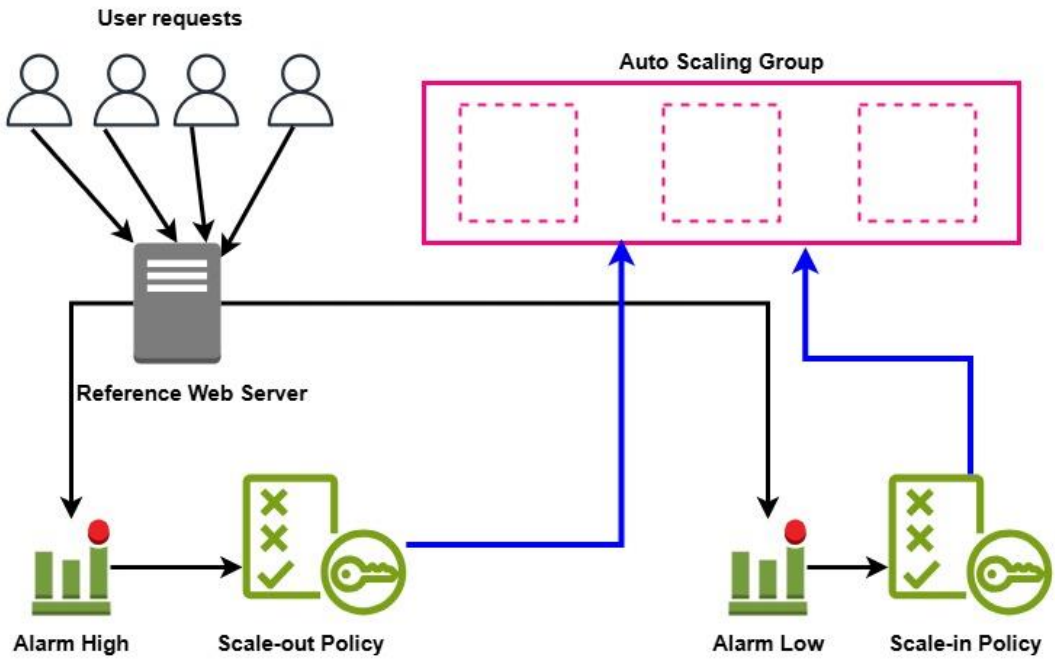

Fig 5.25 Heat-based Autoscaling Implementation

5.4.2 Senlin, ceilometer and Gnocchi

In this approach, the autoscaling solution will be implemented using Ceilometer, Senlin, and Gnocchi. Senlin will be the autoscaling engine. This approach differs from the earlier one; instead of relaying templates, all the configurations like autoscaling group creation, scaling policy creation, and alarm creation, are done in a single template. in this method, a distribution kind of implementation is used. In Heat, an autoscaling group is deployed using hot templates, but in Senlin, a Cluster is created

using a cluster profile written in YAML. Instead of scaling policies used in Heat, Cluster policies are introduced in Senlin. These cluster policies are also created using configuration files written in YAML. Creating the alarms is also done separately in the Senlin implementation. Additionally, there is a new concept in Senlin called Receiver. A receiver is an endpoint that performs actions on a cluster when invoked, It is like an entry point to the system from an external system. In this case, the external system is the Alarming (Aodh) service. This is a webhook-type receiver. Aodh sends a POST request to the webhook once an alarm is triggered, and then, based on the action defined in the receiver, the scaling operation is performed. The step-by-step procedure for implementing this solution is appended in Appendix B. The architecture diagram of this implementation is visualized in Figure 5.26 below.

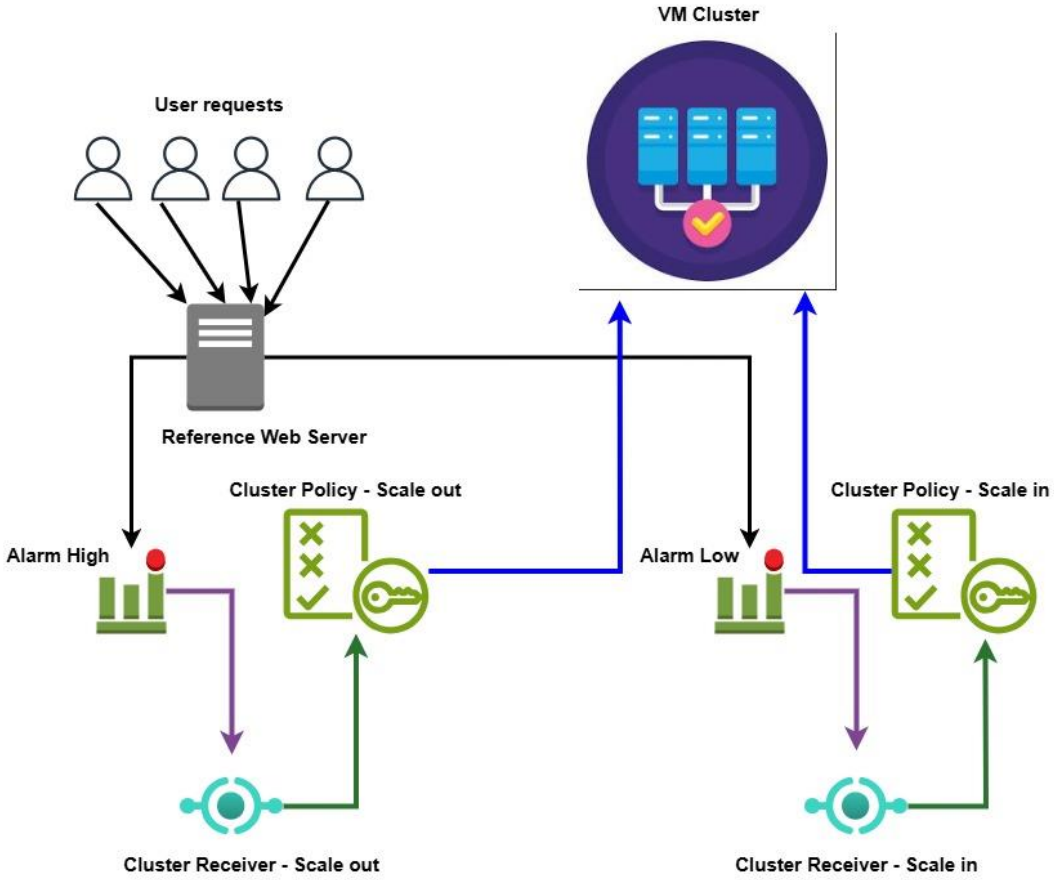

Fig 5.26 Senlin-based Autoscaling Implementation

5.5 Workload pattern generation

To test the autoscaling solutions that were implemented, some workloads are needed. For this purpose, Apache JMeter (version 5.6.3) is used. Ten workload patterns are generated, including five common patterns, and the other five patterns are generated by combining the common workload patterns. These workload patterns and their usability are discussed in detail in section 1.6.2. For example, one of the workload patterns created with JMeter is shown in Figure 5.27. In the X-axis, the workload

applying time period is presented, and the Y-axis shows the number of active threads throughout the workload applying period.

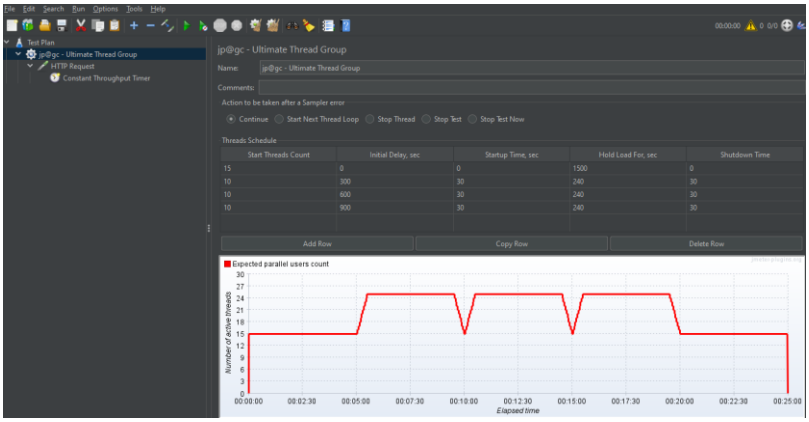

Fig 5.27 Apache JMeter workload generation

Using these 10 workloads, the two autoscaling approaches are tested in an identical environment. Then the outputs are recorded and saved for analysis. The General overall architecture of this implementation is visualized below in Figure 5.28.

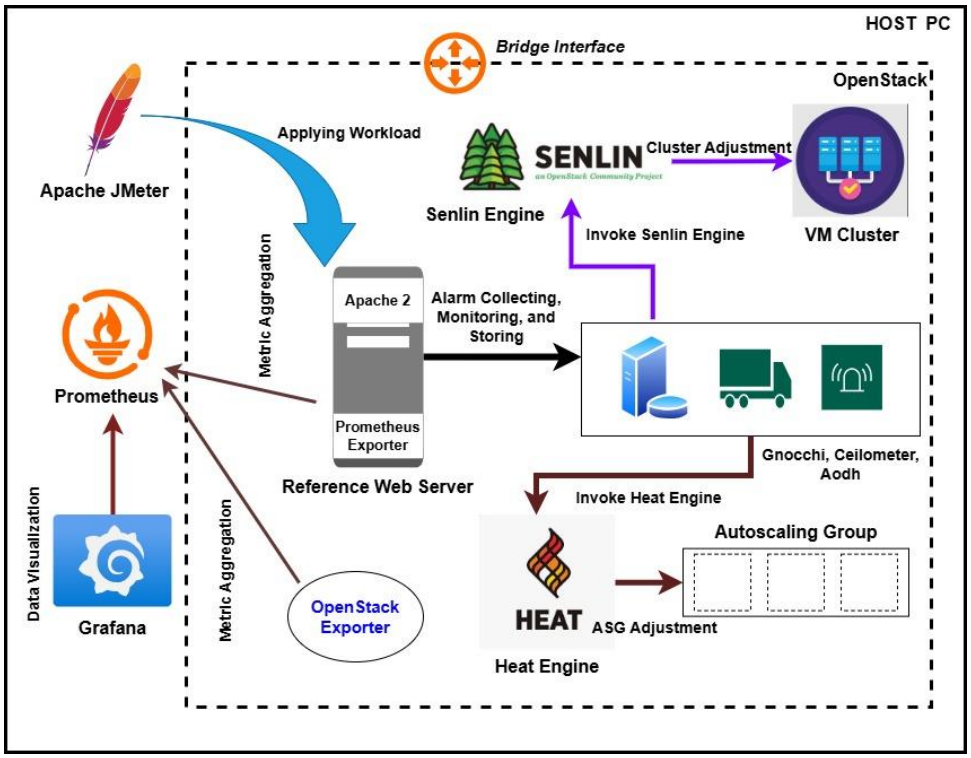

Fig 5.28 The overall architecture of the implementation

CHAPTER 6

RESULTS AND DISCUSSIONS

This section presents the findings from the experiments conducted to evaluate the performance of OpenStack Heat and Senlin as autoscaling mechanisms. The primary objective was to compare their effectiveness in handling dynamic workloads using alarms and scaling triggers, highlighting key metrics such as the number of VMs spawned, VM UP/Downtime durations, Throughput, response times, resource utilization, and adherence to predefined policies, such as cooldown intervals. The experiments used the synthetic workloads generated by benchmarking tools, Apache JMeter, to simulate production scenarios. For each test, identical configurations and workloads were applied to both Heat and Senlin to ensure a fair comparison. Results are analyzed to assess the reliability, efficiency, and scalability of each approach, offering valuable insights into the strengths and limitations of these two autoscaling engines in OpenStack.

Ten workload patterns as mentioned above, are tested using both autoscaling engines. For each test parameters and observations mentioned below are recorded.

- Number of VMs spawned
- VM spawned/delete time
- CPU usage
- Network usage
- Response time for HTTP request
- Throughput of the workload
- Request fail percentage

For the sake of limiting the length of this report, the findings of one test are presented below. We have selected the recorded results for the static workload for the Heat Engine as the reference test. But in this thesis, the same was repeated for Senlin as well. And this whole process is repeated ten times to cover the whole ten workloads.

Fig 6.1 CPU usage during the test1-Heat

Fig 6.2 Network Bandwidth usage during the test1-Heat

Fig 6.3 Behavior of VM creation/deletion during the test1-Heat

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**545174**	**2301**	**0.42%**	**32.92**	**0**	**655**	**14.00**	**79.00**	**97.00**	**133.00**	**454.45**	**4857.79**	**50.82**
HTTP Request	545174	2301	0.42%	32.92	0	655	14.00	79.00	97.00	133.00	454.45	4857.79	50.82

Fig 6.4 Statics of the web requests during the test1-Heat

As mentioned above, all workloads are tested using both Heat and Senlin. A total of 20 tests are carried out during the thesis. The summarization of these tests for both Heat and Senlin is mentioned below.

TABLE 6.1 SUMMARY OF THE RESULTS FOR ALL WORKLOADS WHEN USING HEAT

Test	Workload	Heat Engine									
		No of VM	Test Duration(min)	1st UP	2nd UP	3rd UP	1st Down	2nd Down	3rd Down	Response time(ms)	Throughput(Tras./s)	Fail %
1	Static	3	20	1min 45s	5min 45s	9min 45s	21min 45s	25min 45s	29min 45s	32.92	454.45	0.42
2	High growth	3	20	2min 15s	6min 15s	10min 15s	22min 15s	26min 15s	30min 15s	104.84	472.91	3.82
3	on off	1(3x)	25	2min 05s	12min 20s	22min 05s	7min 05s	17min 05s	27min 05s	31.68	275.88	1.49
4	Aperiodic Bursting	2	20	7min 41s	11m 41s	NA	21min 41s	25min 41s	NA	19.41	356.28	0.05
5	Periodic Bursting	3	25	6min 41s	10min 41s	14min 41s	26min 41s	30min 41s	34min 41s	23.7	403.89	0.15
6	High growth - Static - High Shrink	3	20	6min 03s	10min 03s	14min 03s	21min 03s	25min 03s	29min 03s	25.09	346.56	0.25
7	High growth - Static - High Shrink on off with no interval	2	21	3min 15s	12min 15s	17min 15s	08min 15s	22min 15s	26min 15s	25.2	349.75	0.35
8	High growth - Static - High Shrink on off with interval	1(3x)	31	2min 45s	14min 45s	26min 45s	08min 45s	20min 45s	32min 45s	24.69	237.68	0.25
9	High growth on-off	3	15	1min 35s	5min 35s	9min 35s	16min 35s	20min 35s	24min 35s	104.93	470.23	3.16
10	High growth - High Shrink	2	20	8min 45s	12min 45s	NA	20min 45s	24min 45s	NA	13.79	274.75	0.01

TABLE 6.2 SUMMARY OF THE RESULTS FOR ALL WORKLOADS WHEN USING SENLIN

Test	Workload	Senlin Engine										
		No of VM	Test Duration(min)	1st UP	2nd UP	3rd UP	1st Down	2nd Down	3rd Down	Response time(ms)	Throughput(Tras./s)	Fail %
1	Static	3	20	2min 10s	6min 10s	10min 25s	21min 55s	25min 55s	29min 55s	31.34	462.98	1.81
2	High growth	3	20	2min 05s	5min 50s	9min 50s	21min 50s	25min 50s	29min 50s	102.25	484.6	5.46
3	on off	1(3x)	25	1min 45s	11min 45s	21min 45s	6min 45s	16min 45s	26min 45s	30.46	281.06	2.36
4	Aperiodic Bursting	3	20	7min 25s	11min 25s	15min 25s	21min 25s	25min 25s	29min 25s	19.32	391.96	0.67
5	Periodic Bursting	3	25	6min 50s	10min 50s	14min 50s	26min 50s	30min 50s	34min 50s	23.04	406.37	0.45
6	High growth - Static - High Shrink	3	20	5min 45s	9min 45s	13min 45s	20min 45s	24min 45s	28min 45s	22.93	306.37	0.97
7	High growth - Static - High Shrink on off with no interval	2	21	2min 55s	11min 55s	16min 55s	7min 55s	21min 55s	25min 55s	24.46	350.29	0.34
8	High growth - Static - High Shrink on off with interval	1(3x)	31	2min 50s	14min 50s	26min 50s	7min 50s	19min 50s	31min 50s	24.17	238.32	0.58
9	High growth on-off	3	15	1min 30s	5min 30s	9min 30s	16min 30s	20min 30s	24min 30s	102.79	479.19	4.34
10	High growth - High Shrink	2	20	8min 50s	12min 50s	NA	20min 50s	24min 50s	NA	13.09	276.88	0.12

These results are obtained using identical environments. To determine the best autoscaling engine for each workload pattern, these results need to be compared and analyzed. For this purpose, a Multi-Criteria Decision Analysis (MCDA) decision-making framework is used. MCDA is a well-known systematic approach designed to aid decision-making when multiple, often conflicting criteria must be considered. This is often used in literature. The variant of the Weighted Sum Model (WSM) is used in our analysis. The Weighted Sum Model (WSM) is one of the simplest and most used methods in Multi-Criteria Decision Analysis (MCDA). It involves scoring and weighing different criteria to evaluate.

For results of the Heat and Senlin for each workload are compared and analyzed using MCDA. These results are tabularized and discussed below.

TABLE 6.3 MCDA ANALYSIS OF WORKLOAD 1

Metrics	Weight	Heat	Senlin	Normalized Heat	Normalized Senlin	Weighted Heat	Weighted Senlin
No of VM	0.2	3	3	1	1	0.2	0.2
1st VM up time(s)	0.1	105	130	1	0.807692308	0.1	0.080769231
2nd VM up time(s)	0.1	345	370	1	0.932432432	0.1	0.093243243
3rd VM up time(s)	0.1	585	625	1	0.936	0.1	0.0936
1st VM down time(s)	0.1	1305	1315	1	0.992395437	0.1	0.099239544
2nd VM down time(s)	0.1	1545	1555	1	0.993569132	0.1	0.099356913
3rd VM down time(s)	0.1	1785	1795	1	0.994428969	0.1	0.099442897
Response time(ms)	0.06	32.92	31.34	0.95200486	1	0.057120292	0.06
Throughput (Tras./s)	0.08	454.45	462.98	0.981575878	1	0.07852607	0.08
Fail %	0.06	0.42	1.81	1	0.232044199	0.06	0.013922652
						0.995646362	0.91957448

Table 6.3 presents the MCDA-based evaluation of Heat and Senlin under Workload 1. The comparison was carried out using key autoscaling metrics, each assigned a specific weight reflecting its importance to autoscaling performance. Metrics included the number of VMs, VM up/down times, response time, throughput, and failure rate. In this scenario, both engines spawned the same number of VMs (3), resulting in equal normalized scores for that metric. Heat outperformed Senlin in nearly every VM lifecycle event. It recorded lower values in the 1st, 2nd, and 3rd VM up times (105s, 345s, and 585s, respectively) as well as in all three VM down times. These results suggest that Heat responds more promptly to scaling triggers in this workload, offering faster provisioning and deprovisioning of instances. In terms of response time and throughput, Senlin had a slight advantage. It achieved a lower response time (31.34ms compared to 32.92ms for Heat) and slightly higher throughput (462.98 transactions/sec vs. 454.45). However, these performance gains were marginal and did not offset the higher weight assigned to the more substantial differences observed in the VM operation metrics. For the failure rate, Heat again had the upper hand, showing a significantly lower failure percentage (0.42%) compared to Senlin (1.81%). This reinforces the consistency and reliability of Heat in this workload scenario. Applying the MCDA weights and aggregating the normalized values, Heat achieved a final score of 0.9955, while Senlin scored 0.9196. Therefore, for Workload 1, Heat is determined

to be the more effective autoscaling engine, particularly due to its superior performance in provisioning/deprovisioning operations and lower failure rate.

The results of the MCDA analysis for Workload 2 are presented in Table 6.4. In this workload scenario, the performance of the Heat and Senlin autoscaling engines was evaluated across ten metrics, including VM provisioning times, deprovisioning times, response time, throughput, and failure rate. Each metric was weighted based on its importance to the overall effectiveness of the autoscaling behavior. Senlin demonstrated superior performance across most metrics. Specifically, it recorded faster times in all VM provisioning (1st, 2nd, and 3rd VM up times) and deprovisioning events (1st, 2nd, and 3rd VM down times). These results indicate that Senlin's policy-based engine reacts more swiftly to workload fluctuations compared to Heat's template-driven orchestration. In terms of runtime efficiency, Senlin also exhibited better results with a lower response time (102.25ms vs. 104.84ms) and slightly higher throughput (484.6 transactions/sec vs. 472.91 transactions/sec). These differences, although marginal, further reinforce Senlin’s ability to handle dynamic workloads with higher responsiveness. However, Heat outperformed Senlin in the failure rate metric, showing a significantly lower failure percentage (3.82% vs. 5.46%). This suggests that Heat may offer more reliability under certain conditions, although the impact on overall performance was limited due to the relatively lower weight assigned to this metric. After normalizing and applying the MCDA weighting scheme, Senlin achieved a total weighted score of 0.9819, slightly higher than Heat's score of 0.9588. These results suggest that Senlin is the preferred autoscaling engine for this specific workload pattern, primarily due to its faster scaling operations and superior runtime performance.

TABLE 6.4 MCDA ANALYSIS OF WORKLOAD 2

Metrics	Weight	Heat	Senlin	Normalized Heat	Normalized Senlin	Weighted Heat	Weighted Senlin
No of VM	0.2	3	3	1	1	0.2	0.2
1st VM up time(s)	0.1	135	105	0.777777778	1	0.077777778	0.1
2nd VM up time(s)	0.1	375	350	0.933333333	1	0.093333333	0.1
3rd VM up time(s)	0.1	615	590	0.959349593	1	0.095934959	0.1
1st VM down time(s)	0.1	1335	1310	0.981273408	1	0.098127341	0.1
2nd VM down time(s)	0.1	1575	1550	0.984126984	1	0.098412698	0.1
3rd VM down time(s)	0.1	1815	1790	0.986225895	1	0.09862259	0.1
Response time(ms)	0.06	104.84	102.25	0.975295689	1	0.058517741	0.06
Throughput (Tras./s)	0.08	472.91	484.6	0.975877012	1	0.078070161	0.08
Fail %	0.06	3.82	5.46	1	0.6996337	0.06	0.041978022
						0.958796602	0.981978022

TABLE 6.5 MCDA ANALYSIS OF WORKLOAD 3

Metrics	Weight	Heat	Senlin	Normalized Heat	Normalized Senlin	Weighted Heat	Weighted Senlin
No of VM	0.2	3	3	1	1	0.2	0.2
1st VM up time(s)	0.1	125	105	0.84	1	0.084	0.1
2nd VM up time(s)	0.1	740	705	0.952702703	1	0.09527027	0.1
3rd VM up time(s)	0.1	1325	1305	0.98490566	1	0.098490566	0.1
1st VM down time(s)	0.1	425	405	0.952941176	1	0.095294118	0.1
2nd VM down time(s)	0.1	1025	1005	0.980487805	1	0.09804878	0.1
3rd VM down time(s)	0.1	1625	1605	0.987692308	1	0.098769231	0.1
Response time(ms)	0.06	31.68	30.46	0.961489899	1	0.057689394	0.06
Throughput (Tras./s)	0.08	275.88	281.06	0.981569772	1	0.078525582	0.08
Fail %	0.06	1.49	2.36	1	0.631355932	0.06	0.037881356
						0.966087941	0.977881356

Table 6.5 outlines the MCDA-based evaluation of the Heat and Senlin autoscaling engines under Workload 3. The analysis considers multiple weighted performance metrics, including VM provisioning and deprovisioning times, response time, throughput, and failure rate, all of which collectively influence the overall score. Both engines spawned the same number of VMs, resulting in a tie for the "No of VM" metric. Senlin showed marginally better performance in most of the VM lifecycle operations. It achieved faster startup times in all three VM up events and consistently lower VM down times. For example, the 1st VM up time was 105s for Senlin compared to 125s for Heat, and similarly, the 2nd and 3rd down times were slightly shorter for Senlin. These results suggest that Senlin's policy-based engine handled the workload more efficiently in terms of dynamic resource scaling. Regarding runtime metrics, Senlin again demonstrated favorable performance. It recorded a slightly lower response time (30.46ms vs. 31.68ms) and slightly higher throughput (281.06 vs. 275.88 transactions/sec), highlighting its potential advantage in handling real-time requests and throughput-intensive workloads. However, Heat exhibited a better failure rate, showing a lower fail percentage (1.49%) than Senlin (2.36%). Despite this advantage, the relatively small weight assigned to the fail rate metric limited its influence on the final score. When all metrics were normalized and weighted using MCDA, Senlin achieved a total score of 0.9779, slightly surpassing Heat's score of 0.9661. This indicates that Senlin is the preferred autoscaling engine for Workload 3, primarily due to its consistently faster scaling behavior and improved runtime performance, even though it demonstrated a slightly higher failure rate.

Table 6.6 outlines the MCDA-based evaluation of the Heat and Senlin autoscaling engines under Workload 4. The analysis considers multiple weighted performance metrics, including VM provisioning and deprovisioning times, response time, throughput, and failure rate, all of which collectively influence the overall score. Senlin engines assigned 3 VMs while Heat assigned only 2 VMs. Senlin showed marginally better performance in most of the VM lifecycle operations. It achieved faster startup times in all three VM up events and consistently lower VM down times.

For example, the 1st VM up time was 445s for Senlin compared to 461s for Heat. These results suggest that Senlin's policy-based engine handled the workload more efficiently in terms of dynamic resource scaling. Regarding runtime metrics, Senlin again demonstrated favorable performance. It recorded a slightly lower response time (19.32ms vs. 19.41ms) and slightly higher throughput (391.96 vs. 356.28 transactions/sec), highlighting its potential advantage in handling real-time requests and throughput-intensive workloads. However, Heat exhibited a better failure rate, showing a lower fail percentage (0.05%) than Senlin (0.67%). Despite this advantage, the relatively small weight assigned to the fail rate metric limited its influence on the final score. When all metrics were normalized and weighted using MCDA, Senlin achieved a total score of 0.9445, surpassing Heat's score of 0.7178. This indicates that Senlin is the preferred autoscaling engine for Workload 4, primarily due to its consistently faster scaling behavior and improved runtime performance, even though it demonstrated a slightly higher failure rate.

TABLE 6.6 MCDA ANALYSIS OF WORKLOAD 4

Metrics	Weight	Heat	Senlin	Normalized Heat	Normalized Senlin	Weighted Heat	Weighted Senlin
No of VM	0.2	2	3	0.666666667	1	0.133333333	0.2
1st VM up time(s)	0.1	461	445	0.965292842	1	0.096529284	0.1
2nd VM up time(s)	0.1	701	685	0.977175464	1	0.097717546	0.1
3rd VM up time(s)	0.1		925	0	1	0	0.1
1st VM down time(s)	0.1	1301	1285	0.987701768	1	0.098770177	0.1
2nd VM down time(s)	0.1	1541	1525	0.989617132	1	0.098961713	0.1
3rd VM down time(s)	0.1		1,765	0	1	0	0.1
Response time(ms)	0.06	19.41	19.32	0.995363215	1	0.059721793	0.06
Throughput (Tras./s)	0.08	356.28	391.96	0.908970303	1	0.072717624	0.08
Fail %	0.06	0.05	0.67	1	0.074626866	0.06	0.004477612
						0.717751471	0.944477612

Table 6.7 presents the MCDA-based evaluation of Heat and Senlin under Workload 5. The comparison was carried out using key autoscaling metrics, each assigned a specific weight reflecting its importance to autoscaling performance. Metrics included the number of VMs, VM up/down times, response time, throughput, and failure rate. In this scenario, both engines spawned the same number of VMs (3), resulting in equal normalized scores for that metric. Heat outperformed Senlin in nearly every VM lifecycle event. It recorded lower values in the 1st, 2nd, and 3rd VM up times (401s, 641s, and 881s, respectively) as well as in all three VM down times. These results suggest that Heat responds more promptly to scaling triggers in this workload, offering faster provisioning and deprovisioning of instances. In terms of response time and throughput, Senlin had a slight advantage. It achieved a lower response time (23.04ms compared to 23.70ms for Heat) and slightly higher throughput (406.37 transactions/sec vs. 403.89). However, these performance gains were marginal and did not offset the higher weight assigned to the more substantial differences observed in the VM

operation metrics. For the failure rate, Heat again had the upper hand, showing a significantly lower failure percentage (0.15%) compared to Senlin (0.45%). This reinforces the consistency and reliability of Heat in this workload scenario. Applying the MCDA weights and aggregating the normalized values, Heat achieved a final score of 0.9978, while Senlin scored 0.9539. Therefore, for Workload 5, Heat is determined to be the more effective autoscaling engine, particularly due to its superior performance in provisioning/deprovisioning operations and lower failure rate.

TABLE 6.7 MCDA ANALYSIS OF WORKLOAD 5

Metrics	Weight	Heat	Senlin	Normalized Heat	Normalized Senlin	Weighted Heat	Weighted Senlin
No of VM	0.2	3	3	1	1	0.2	0.2
1st VM up time(s)	0.1	401	410	1	0.97804878	0.1	0.097804878
2nd VM up time(s)	0.1	641	650	1	0.986153846	0.1	0.098615385
3rd VM up time(s)	0.1	881	890	1	0.98988764	0.1	0.098988764
1st VM down time(s)	0.1	1601	1610	1	0.994409938	0.1	0.099440994
2nd VM down time(s)	0.1	1841	1850	1	0.995135135	0.1	0.099513514
3rd VM down time(s)	0.1	2081	2090	1	0.99569378	0.1	0.099569378
Response time(ms)	0.06	23.7	23.04	0.972151899	1	0.058329114	0.06
Throughput (Tras./s)	0.08	403.89	406.37	0.993897187	1	0.079511775	0.08
Fail %	0.06	0.15	0.45	1	0.333333333	0.06	0.02
						0.997840889	0.953932912

TABLE 6.8 MCDA ANALYSIS OF WORKLOAD 6

Metrics	Weight	Heat	Senlin	Normalized Heat	Normalized Senlin	Weighted Heat	Weighted Senlin
No of VM	0.2	3	3	1	1	0.2	0.2
1st VM up time(s)	0.1	363	345	0.950413223	1	0.095041322	0.1
2nd VM up time(s)	0.1	603	585	0.970149254	1	0.097014925	0.1
3rd VM up time(s)	0.1	843	825	0.978647687	1	0.097864769	0.1
1st VM down time(s)	0.1	1263	1245	0.985748219	1	0.098574822	0.1
2nd VM down time(s)	0.1	1503	1485	0.988023952	1	0.098802395	0.1
3rd VM down time(s)	0.1	1743	1725	0.989672978	1	0.098967298	0.1
Response time(ms)	0.06	25.09	22.93	0.913909924	1	0.054834595	0.06
Throughput (Tras./s)	0.08	346.56	306.37	1	0.884031625	0.08	0.07072253
Fail %	0.06	0.25	0.97	1	0.257731959	0.06	0.015463918
						0.981100127	0.946186448

The MCDA evaluation for Workload 6 is presented in Table 6.8, comparing Heat and Senlin across ten key metrics. These metrics include VM startup and shutdown times, response time, throughput, and failure rate, with weights assigned to reflect their

relative importance to autoscaling performance. In this workload, both Heat and Senlin were assigned the same number of virtual machines (3), resulting in equal scores for that metric. Senlin exhibited faster startup times for all three VM provisioning stages. For instance, the 1st VM up time for Senlin was 345s compared to 363s for Heat. Similarly, the 2nd and 3rd VM up times were shorter in Senlin, earning full normalized scores for each. Senlin also showed consistent advantages in VM down time, performing better across all three deprovisioning events. These faster responses in scaling operations reflect Senlin's more agile and policy-driven approach to autoscaling. In terms of response time, Senlin again outperformed Heat with a significantly lower average response time (22.93ms vs. 25.09ms). However, Heat maintained a clear advantage in throughput, achieving 346.56 transactions/sec compared to Senlin's 306.37, which translated to a full normalized score for Heat and a slightly penalized score for Senlin. Failure rate analysis also favored Heat. Heat reported a minimal failure rate of 0.25% versus Senlin's 0.97%, earning the maximum normalized value for this metric. This highlights Heat's potential reliability in maintaining system stability under certain workload stress conditions. When all the weighted metrics were aggregated, Heat scored a total of 0.9811, surpassing Senlin's score of 0.9462. Based on this analysis, Heat is identified as the better-performing autoscaling engine for Workload 6. Its superior throughput and reliability metrics played a crucial role in outweighing Senlin's speed advantages in provisioning and response time.

TABLE 6.9 MCDA ANALYSIS OF WORKLOAD 7

Metrics	Weight	Heat	Senlin	Normalized Heat	Normalized Senlin	Weighted Heat	Weighted Senlin
No of VM	0.2	2	2	1	1	0.2	0.2
1st VM up time(s)	0.1	195	175	0.897435897	1	0.08974359	0.1
2nd VM up time(s)	0.1	735	715	0.972789116	1	0.097278912	0.1
3rd VM up time(s)	0.1	1035	1015	0.980676329	1	0.098067633	0.1
1st VM down time(s)	0.1	495	475	0.95959596	1	0.095959596	0.1
2nd VM down time(s)	0.1	1335	1315	0.985018727	1	0.098501873	0.1
3rd VM down time(s)	0.1	1575	1555	0.987301587	1	0.098730159	0.1
Response time(ms)	0.06	25.2	24.46	0.970634921	1	0.058238095	0.06
Throughput (Tras./s)	0.08	349.75	350.29	0.99845842	1	0.079876674	0.08
Fail %	0.06	0.35	0.34	0.971428571	1	0.058285714	0.06
						0.974682245	1

Table 6.9 presents the MCDA-based comparison of the Heat and Senlin autoscaling engines under Workload 7. This scenario generated fewer VMs (2), with performance assessed across key metrics such as provisioning/deprovisioning times, response time, throughput, and failure percentage. Both Heat and Senlin received identical scores for the number of VMs generated, which contributed equally to the weighted score. Senlin exhibited consistently better performance across all provisioning (up time) and deprovisioning (down time) metrics. For example, the 1st VM up time for Senlin was

175s compared to 195s for Heat, and similar advantages were noted in the 2nd and 3rd up/down times. These improvements in lifecycle timing demonstrate Senlin's efficiency in handling dynamic workloads with quicker scaling actions. Senlin also showed marginally better runtime efficiency. Its response time was slightly lower (24.46ms vs. 25.2ms), and it achieved slightly higher throughput (350.29 vs. 349.75 transactions/sec). While the absolute differences were minimal, Senlin consistently earned full normalized scores for these metrics, giving it a measurable edge. In terms of reliability, Senlin once again outperformed Heat, reporting a lower failure rate (0.34% compared to 0.35%). Though the difference is small, it contributes positively to the final evaluation. When all the normalized and weighted scores were combined, Senlin achieved a perfect score of 1.0000, whereas Heat recorded 0.9747. These results indicate that Senlin is the most effective autoscaling engine for Workload 7. Its consistent advantages in provisioning speed, runtime responsiveness, and reliability metrics make it better suited for workloads of this nature.

TABLE 6.10 MCDA ANALYSIS OF WORKLOAD 8

Metrics	Weight	Heat	Senlin	Normalized Heat	Normalized Senlin	Weighted Heat	Weighted Senlin
No of VM	0.2	3	3	1	1	0.2	0.2
1st VM up time(s)	0.1	165	170	1	0.970588235	0.1	0.097058824
2nd VM up time(s)	0.1	885	890	1	0.994382022	0.1	0.099438202
3rd VM up time(s)	0.1	1605	1610	1	0.99689441	0.1	0.099689441
1st VM down time(s)	0.1	525	470	0.895238095	1	0.08952381	0.1
2nd VM down time(s)	0.1	1245	1190	0.955823293	1	0.095582329	0.1
3rd VM down time(s)	0.1	1965	1910	0.972010178	1	0.097201018	0.1
Response time(ms)	0.06	24.69	24.17	0.978938842	1	0.05873633	0.06
Throughput (Tras./s)	0.08	237.68	238.32	0.997314535	1	0.079785163	0.08
Fail %	0.06	0.25	0.58	1	0.431034483	0.06	0.025862069
						0.98082865	0.962048536

Table 6.10 illustrates the MCDA analysis results for Heat and Senlin under Workload 8. This workload scenario generated 3 VMs in both engines, with evaluation metrics covering scaling performance, system responsiveness, throughput, and reliability. Equal weighting was assigned for the number of VMs generated, contributing identically to the normalized score. In terms of VM provisioning, Heat demonstrated marginally better performance. It achieved lower times for all three up-time stages, scoring the maximum normalized value of 1.0 in each case. Senlin, while close in performance, scored slightly lower in all three startup events. For example, the 1st VM up time for Senlin was 170s compared to 165s for Heat, resulting in a normalized score of 0.9706. Senlin, however, outperformed Heat in all downtime events. It recorded quicker deprovisioning for the 1st, 2nd, and 3rd VM shutdowns, achieving normalized values of 1.0 for each. Heat's shutdown times were longer, with normalized values ranging from 0.895 to 0.972. These results suggest that Senlin's policy-driven control led to faster resource release during workload scale-down. Runtime performance metrics further reinforced Senlin's strengths. It achieved a slightly lower response time (24.17ms vs. 24.69ms) and slightly higher throughput (238.32 vs. 237.68

transactions/sec). These small differences contributed positively to Senlin's performance evaluation. However, Heat demonstrated greater reliability with a significantly lower failure rate of 0.25%, compared to 0.58% for Senlin. Consequently, Heat received the full normalized score for this metric, whereas Senlin's score dropped to 0.431. When combining all weighted and normalized values using MCDA, Heat achieved a total score of 0.9808, while Senlin scored 0.9620. While both engines performed well, Heat emerged as the more balanced and reliable autoscaling engine for Workload 8, particularly due to its stronger provisioning efficiency and lower fail rate, which compensated for Senlin's better shutdown times and slightly faster runtime behavior.

TABLE 6.11 MCDA ANALYSIS OF WORKLOAD 9

Metrics	Weight	Heat	Senlin	Normalized Heat	Normalized Senlin	Weighted Heat	Weighted Senlin
No of VM	0.2	3	3	1	1	0.2	0.2
1st VM up time(s)	0.1	95	90	0.947368421	1	0.094736842	0.1
2nd VM up time(s)	0.1	335	330	0.985074627	1	0.098507463	0.1
3rd VM up time(s)	0.1	575	570	0.991304348	1	0.099130435	0.1
1st VM down time(s)	0.1	995	990	0.994974874	1	0.099497487	0.1
2nd VM down time(s)	0.1	1235	1230	0.995951417	1	0.099595142	0.1
3rd VM down time(s)	0.1	1475	1470	0.996610169	1	0.099661017	0.1
Response time(ms)	0.06	104.93	102.79	0.979605451	1	0.058776327	0.06
Throughput (Tras./s)	0.08	470.23	479.19	0.98130178	1	0.078504142	0.08
Fail %	0.06	3.16	4.34	1	0.728110599	0.06	0.043686636
						0.988408855	0.983686636

Table 6.11 provides the MCDA evaluation of the Heat and Senlin autoscaling engines under Workload 9. The comparison focused on various autoscaling performance indicators, including No of VMs, startup and shutdown times, response time, throughput, and failure percentage. Senlin outperformed Heat in all 9 scaling operation metrics, including VM up time and VM down time. Though the differences were minor, Senlin recorded consistently faster times across provisioning and deprovisioning events. For instance, the 1st VM up time for Senlin was 90s versus 95s for Heat, and similar margins were observed in all subsequent scale actions. As a result, Senlin received full normalized scores (1.0) for all these timing-related metrics, while Heat received slightly lower values. In terms of runtime performance, Senlin again held the advantage. Its response time was slightly faster at 102.79ms compared to Heat's 104.93ms, and its throughput was marginally higher (479.19 vs. 470.23 transactions/sec). These differences, while not dramatic, contributed positively to Senlin's overall efficiency rating. However, in the area of system reliability, Heat delivered superior results. It recorded a lower failure rate of 3.16%, compared to

Senlin's 4.34%. Consequently, Heat received the full normalized score for this metric, while Senlin was penalized with a reduced score of 0.728.

When all weighted and normalized scores were aggregated using the MCDA method, Heat achieved a final score of 0.9884, slightly higher than Senlin's 0.9837. Despite Senlin's dominance in scaling speed and runtime performance, Heat was marginally the better-performing engine for Workload 9, primarily due to its stronger reliability score. This highlights that even small differences in failure rates can have a meaningful impact in autoscaling environments where consistency and uptime are critical.

TABLE 6.12 MCDA ANALYSIS OF WORKLOAD 10

Metrics	Weight	Heat	Senlin	Normalized Heat	Normalized Senlin	Weighted Heat	Weighted Senlin
No of VM	0.2	2	2	1	1	0.2	0.2
1st VM up time(s)	0.15	525	530	1	0.990566038	0.15	0.148584906
2nd VM up time(s)	0.15	765	770	1	0.993506494	0.15	0.149025974
1st VM down time(s)	0.15	1245	1250	1	0.996	0.15	0.1494
2nd VM down time(s)	0.15	1485	1490	1	0.996644295	0.15	0.149496644
Response time(ms)	0.06	13.79	13.09	0.949238579	1	0.056954315	0.06
Throughput (Tras./s)	0.08	274.75	276.88	0.992307137	1	0.079384571	0.08
Fail %	0.06	0.01	0.12	1	0.083333333	0.06	0.005
						0.996338886	0.941507524

Table 6.12 presents the MCDA-based evaluation of the Heat and Senlin autoscaling engines under Workload 10. Both Heat and Senlin scaled up to a total of two virtual machines. In terms of VM provisioning and deprovisioning, Heat demonstrated slightly better performance across all lifecycle operations. It recorded faster startup times for both VMs and quicker shutdown times, achieving the maximum normalized score of 1.0 for each of these four metrics. Senlin trailed only slightly in each case, with normalized values very close to 1.0, indicating marginal but consistent differences in favor of Heat. Senlin outperformed Heat in runtime responsiveness, achieving a lower response time (13.09ms compared to Heat's 13.79ms), which translated into a full normalized score for this metric. However, the Heat still performed well, earning a score of approximately 0.949 for response time. Throughput values were also slightly higher for Senlin (276.88 vs. 274.75 transactions/sec), giving it a full normalized score of 1.0 compared to Heat's 0.992. While these differences were small, they reflected Senlin's advantage in runtime throughput and latency under this workload. However, Heat displayed a significantly better failure rate, with only 0.01% compared to Senlin's 0.12%. This large discrepancy led to a normalized score of 1.0 for Heat and a much lower score of 0.083 for Senlin for the failure percentage metric. Given that reliability is often a critical factor in production environments, this outcome notably impacted the final evaluation. When all metrics were normalized,

weighted, and aggregated using the MCDA method, Heat obtained a final score of 0.9963, while Senlin achieved 0.9415. Therefore, Heat was identified as the superior autoscaling engine for Workload 10, primarily due to its faster lifecycle operation times and significantly higher reliability, which outweighed Senlin's slight advantages in runtime performance.

TABLE 6.13 THE BETTER SOLUTION FOR EACH WORKLOAD BASED ON MCDA ANALYSIS

Test	Workload	Winner
1	Static	Heat
2	High growth	Senlin
3	on off	Senlin
4	Aperiodic Bursting	Senlin
5	Periodic Bursting	Heat
6	High growth - Static - High Shrink	Heat
7	High growth - Static - High Shrink on off with no interval	Senlin
8	High growth - Static - High Shrink on off with interval	Heat
9	High growth on-off	Heat
10	High growth - High Shrink	Heat

CHAPTER 7

CONCLUSION AND FUTURE WORK

This study evaluated the autoscaling capabilities of OpenStack Heat and Senlin, focusing on their performance under dynamic workloads. Both solutions were assessed using identical configurations, simulated benchmarking, and real-world HTTP workload patterns to ensure a comprehensive and fair comparison. This research extensively compared the two solutions' efficiency, scalability, and adherence to policies such as alarms and cooldown intervals. The findings indicate that while both Heat and Senlin can effectively handle autoscaling tasks, Senlin showed better performance compared to Heat for the following workloads,

- High growth
- on and off
- Aperiodic Bursting
- High growth - Static - High Shrink on-off with no interval

On the other hand, Heat performed well with the workloads below compared to Senlin.

- Static
- Periodic Bursting
- High growth - Static - High Shrink
- High growth - Static - High Shrink on off with interval
- High growth on-off
- High growth - High Shrink

If we generalize the above results, we can conclude that Senlin offers significant advantages in environments with highly dynamic and unpredictable workloads. Its advanced policy enforcement, such as flexible placement strategies and rapid response to alarms via webhooks, enables more efficient resource utilization and cluster stability. On the other hand, with its stack-based approach, Heat remains a robust option for environments with less dynamic and predictive autoscaling needs.

This study contributes to the OpenStack community by offering a detailed analysis of two of its key autoscaling mechanisms. These insights can help OpenStack operators, developers, and architects make informed decisions when designing autoscaling solutions tailored to specific workloads. The evaluation of HTTP workload patterns also adds practical value, providing a benchmark that reflects realistic use cases. Moreover, the methodologies and findings of this study can serve as a foundation for further research and innovation in the OpenStack ecosystem. For example, they can guide the development of improved autoscaling policies, deeper integrations with monitoring tools like Prometheus, and optimizations for scenarios such as multi-region deployments or stateful applications.

Ultimately, this research underscores the importance of aligning autoscaling strategies with workload characteristics and operational goals. By fostering a better

understanding of the strengths and limitations of Heat and Senlin, this work supports the OpenStack community's ongoing efforts to build efficient, resilient, and scalable cloud platforms.

REFERENCES

[1] S. Chouliaras and S. Sotiriadis, “An adaptive auto-scaling framework for cloud resource provisioning,” Futur. Gener. Comput. Syst., vol. 148, pp. 173–183, 2023, doi: 10.1016/j.future.2023.05.017.

[2] H. Arabnejad, P. Jamshidi, G. Estrada, N. El Ioini, and C. Pahl, “An auto-scaling cloud controller using fuzzy Q-learning - Implementation in OpenStack,” Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 9846 LNCS, no. September, pp. 152–167, 2016, doi: 10.1007/978-3-319-44482-6_10.

[3] S. Ranger, “What is cloud computing? Everything you need to know about the cloud explained,” *ZDNet*, Feb. 25, 2022. https://www.zdnet.com/article/what-is-cloud-computing-everything-you-need-to-know-about-the-cloud/ (visited on 31-12-2023).

[4] Amazon, “Types of cloud computing,” Amazon Web Services, Inc., 2019. https://aws.amazon.com/types-of-cloud-computing/ (visited on 31-12-2023).

[5] B. L. Sahu and R. Tiwari, "A Comprehensive Study on Cloud Computing," Int. J., vol. 2, no. 9, 2012.

[6] W. Lau, “An Introduction to Cloud Computing Characteristics and Service/Deployment Models Cloud Zone,” 16-May-2012. [Online]. Available: http://cloud.dzone.com/articles/introduction cloud-Computing.

[7] W. Voorsluys, J. Broberg, and R. Buyya, “Introduction to Cloud Computing,” in Cloud Computing, R. Buyya, J. Broberg, and A. Goscinski, Eds. John Wiley & Sons, Inc., 2011, pp. 1– 41.

[8] GeeksforGeeks, “Cloud Deployment Models,” GeeksforGeeks, Jul. 11, 2021. https://www.geeksforgeeks.org/cloud-deployment-models/ (visited on 31-12-2023).

[9] M. I. Hossain, "Dynamic Scaling of a Web-Based Application in a Cloud Architecture," 2014.

[10] “Virtualization in Cloud Computing - javatpoint,” www.javatpoint.com, 2011. https://www.javatpoint.com/virtualization-in-cloud-computing. (visited on 31-12-2023).

[11] A. Aziz and W. Wagdy, "Auto Scaling Solutions for Cloud Applications," International Journal of Knowledge and Systems Science, vol. 24, 2023.

[12] S. Brenner, B. Garbers, and R. Kapitza, "Adaptive and Scalable High Availability for Infrastructure Clouds," Distributed Applications and Interoperable Systems, Lecture Notes in Computer Science, 2014, pp. 16-30.

[13] P. R. Gupta, S. Taneja, and A. Datt, "Using Heat and Ceilometer for providing Autoscaling in OpenStack," 2014.

[14] L. R. Moore, K. Bean, and T. Ellahi, "A Coordinated Reactive and Predictive Approach to Cloud Elasticity," in The Fourth International Conference on Cloud Computing, GRIDs, and Virtualization, 2013, pp. 87–92.

[15] T. Lorido-Botrán, J. Miguel-Alonso, and J. A. Lozano, "Auto-scaling Techniques for Elastic Applications in Cloud Environments," Journal of Grid Computing, vol. 12, pp. 559–592, Sept. 2012.

[16] A. Beloglazov and R. Buyya, "Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers," in The 8th International Workshop on Middleware for Grids, Clouds e-Sci, 2010, p. 4.

[17] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, "Using Reinforcement Learning for Autonomic Resource Allocation in Clouds: towards a fully automated workflow," in Seventh International Conference on Autonomic and Autonomous Systems, May 2011, pp. 67-74.

[18] P. Jamshidi, A. Ahmad, and C. Pahl, "Autonomic resource provisioning for cloud-based software," in SEAMS, 2014, pp. 95–104.

[19] P. Jamshidi, A. Sharifloo, and C. Pahl, "Fuzzy Self-Learning Controllers for Elasticity Management in Dynamic Cloud Architectures," July 2016.

[20] P. Y. Glorennec and L. Jouffe, "Fuzzy Q-learning," in The 6th International Fuzzy Systems Conference, Vol. 2, IEEE, 1997.

[21] E. G. Radhika and G. Sudha Sadasivam, "A review on prediction-based autoscaling techniques for heterogeneous applications in cloud environment," in Materials Today: Proceedings, vol. 45, 2021, pp. 2793-2800.

[22] A. Bankole and S. Ajila, "Cloud client prediction models for cloud resource provisioning in a multitier web application environment," in IEEE 7th International Symposium on Service-Oriented System Engineering (SOSE), March 2013, pp. 156–161.

[23] S. Ajila and A. Bankole, "Cloud client prediction models using machine learning technique," in IEEE 37th Annual Computer Software and Applications Conference (COMPSAC), July 2013, pp. 134–142.

[24] A. Bankole and S. Ajila, "Predicting cloud resource provisioning using machine learning techniques," in 26th Annual IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), May 2013, pp. 1–4.

[25] J. S. Felix A. Gers and N. N. Schraudolph, "Learning precise timing with LSTM recurrent networks," in Journal of Machine Learning Research, vol. 3, 2002, pp. 115–143.

[26] S. Kardani Moghaddam, R. Buyya, and K. Ramamohanarao, "ACAS: An anomaly-based cause-aware auto-scaling framework for clouds," 2019.

[27] X. Dutreilh, N. Rivierre, A. Moreau, J. Malenfant, and I. Truck, "From data center resource allocation to control theory and back," in Intl Conf on Cloud Computing (CLOUD), 2010, pp. 410–417.

[28] R. Han, L. Guo, M. M. Ghanem, and Y. Guo, "Lightweight resource scaling for cloud applications," in Cluster, Cloud and Grid Computing (CCGrid), 2012, pp. 644–651.

[29] M. Z. Hasan, E. Magana, A. Clemm, L. Tucker, and S. L. D. Gudreddi, "Integrated and autonomic cloud resource scaling," in Network Operations and Management Symposium (NOMS), 2012, pp. 1327–1334.

[30] T. C. Chieu, A. Mohindra, and A. A. Karve, "Scalability and performance of web applications in a compute cloud," in Intl Conf on e-Business Engineering, 2011.

[31] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "A hybrid reinforcement learning approach to autonomic resource allocation," in Intl Conf on Autonomic Computing, 2006, pp. 65–73.

[32] J. Rao, X. Bu, C. Z. Xu, L. Wang, and G. Yin, "Vconf: a reinforcement learning approach to virtual machines auto-configuration," in Intl Conference on Autonomic Computing, 2009, pp. 137–146.

[33] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in Network Operations and Management Symposium (NOMS), 2012, pp. 204–212.

[34] P. Padala, K. Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant, "Automated control of multiple virtualized resources," in Europ Conf on Computer systems, 2009, pp. 13–26.

[35] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. Jordan, and D. Patterson, "Statistical machine learning makes automatic control practical for internet data centers," in HotCloud'09: Proceedings of the Workshop on Hot Topics in Cloud Computing, 2009.

[36] H. Liu and S. Wee, "Web server farm in the cloud: Performance evaluation and dynamic architecture," in CloudCom '09: Proceedings of the 1st International Conference on Cloud Computing, 2009, pp. 369–380.

[37] M. T. Krieger, O. Torreno, O. Trelles, and D. Kranzlmüller, "Building an open-source cloud environment with auto-scaling resources for executing bioinformatics and biomedical workflows," Future Generation Computer Systems, vol. 67, pp. 329–340, 2017.

[38] V. A. Farias, F. R. Sousa, J. G. R. Maia, J. P. P. Gomes, and J. C. Machado, "Regression-based performance modeling and provisioning for NoSQL cloud databases," Future Generation Computer Systems, vol. 79, pp. 72–81, 2018.

[39] M. Wajahat, A. Karve, A. Kochut, and A. Gandhi, "Mlscale: A machine learning-based application-agnostic autoscaler," Sustainable Computing: Informatics and Systems, 2017, 2017.

[40] V. Persico, D. Grimaldi, A. Pescapè, A. Salvi, and S. Santini, "A fuzzy approach based on heterogeneous metrics for scaling out public clouds," IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 8, pp. 2117–2130, Aug 2017.

[41] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. V. Papadopoulos, B. Ghit, D. Epema, and A. Iosup, "An experimental performance evaluation of autoscaling policies for complex workflows," in Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, 2017, pp. 75–86.

[42] K. Salah, P. Calyam, and R. Boutaba, "Analytical model for elastic scaling of cloud-based firewalls," IEEE Transactions on Network and Service Management, vol. 14, no. 1, pp. 136–146, 2017.

[43] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," Journal of Grid Computing, vol. 12, no. 4, pp. 559–592, 2014.

[44] B. Jennings and R. Stadler, "Resource management in clouds: Survey and research challenges," Journal of Network and Systems Management, vol. 23, no. 3, pp. 567–619, 2015.

[45] S. Wu, B. Li, X. Wang, and H. Jin, "Hybridscaler: Handling bursting workload for multi-tier web applications in the cloud," in Parallel and Distributed Computing (ISPDC), 2016, pp. 141–148.

[46] C. Qu, R. N. Calheiros, and R. Buyya, "A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances," Journal of Network and Computer Applications, vol. 65, pp. 167–180, 2016.

[47] RunAI. "Machine Learning in the Cloud." [Online]. Available: https://www.run.ai/guides/machine-learning-in-the-cloud. (visited on 08-01-2024).

[48] Columbus Global. "How to Deploy and Support Trained AI and ML Models." [Online]. Available: https://www.columbusglobal.com/en/blog/how-to-deploy-and-support-trained-ai-and-ml-models. (visited on 08-01-2024).

[49] ScienceDirect. "Performance prediction in dynamic clouds using transfer learning." [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0743731522001307. (visited on 08-01-2024).

[50] F. Moradi, R. Stadler, and A. Johnsson, "Performance prediction in dynamic clouds using transfer learning," in 2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), IEEE, 2019.

[51] Z. Ahamed et al., "Technical Study of Deep Learning in Cloud Computing for Accurate Workload Prediction," Electronics, vol. 12, no. 3, 2023, p. 650.

[52] InfoWorld. "Downsides to Using Cloud Autoscaling Systems." [Online]. Available: https://www.infoworld.com/article/3702111/downsides-to-using-cloud-autoscaling-systems.html. (visited on 08-01-2024).

[53] Logicworks. "Common Mistakes and Misconceptions about Auto Scaling in AWS." [Online]. Available: https://www.logicworks.com/blog/2017/12/common-mistakes-misconceptions-auto-scaling-aws/. (visited on 08-01-2024).

[54] Alibaba Cloud. "Challenges and Considerations about Alibaba Cloud Application Scaling." [Online]. Available: https://www.alibabacloud.com/blog/challenges-and-considerations-about-alibaba-cloud-application-scaling_597496. (visited on 08-01-2024).

[55] InfoWorld. "Downsides to Using Cloud Autoscaling Systems." [Online]. Available: https://www.infoworld.com/article/3702111/downsides-to-using-cloud-autoscaling-systems.html. (visited on 08-01-2024).

[56] Datadog. "Auto Scaling." [Online]. Available: https://www.datadoghq.com/knowledge-center/auto-scaling/. (visited on 08-01-2024).

[57] Nutanix. "Top 11 Hard-Won Lessons Learned about AWS Auto Scaling." [Online]. Available: https://nutanix.medium.com/top-11-hard-won-lessons-learned-about-aws-auto-scaling-5bfe56da755f. (visited on 08-01-2024).

[58] AWS Amazon. "Auto Scaling." [Online]. Available: https://aws.amazon.com/autoscaling/. (visited on 08-01-2024).

[59] Wallarm. "What Is Auto Scaling?" [Online]. Available: https://www.wallarm.com/what/what-is-auto-scaling. (visited on 08-01-2024).

[60] TechTarget. "Autoscaling." [Online]. Available: https://www.techtarget.com/searchcloudcomputing/definition/autoscaling. (visited on 08-01-2024).

[61] AWS Amazon. "Auto Scaling Benefits." [Online]. Available: https://docs.aws.amazon.com/autoscaling/ec2/userguide/auto-scaling-benefits.ht. (visited on 08-01-2024).

[62] A. Solano, R. Dormido, N. Duro, and J. Sánchez, "A Self-Provisioning Mechanism in OpenStack for IoT Devices," Sensors, vol. 16, 2016, p. 1306. DOI: 10.3390/s16081306.

[63] Huawei. "Ceilometer Telemetry in OpenStack." [Online]. Available: https://forum.huawei.com/enterprise/en/ceilometer-telemetry-in-openstack/thread/673921301179416576-667213860102352896. (visited on 02-09-2024).

[64] STFC Cloud Docs. "Monitoring VMs with Aodh and Gnocchi." [Online]. Available: https://stfc-cloud-docs.readthedocs.io/en/latest/Aodh-and-Gnocchi/MonitoringVMsAodhGnocchi.html. (visited on 02-09-2024).

[65] Red Hat. "Configuring the Time Series Database Gnocchi for Telemetry." [Online]. Available: https://docs.redhat.com/en/documentation/red_hat_openstack_platform/16.1/html/log

ging_monitoring_and_troubleshooting_guide/configuring_the_time_series_database_gnocchi_for_telemetry#configuring_the_time_series_database_gnocchi_for_telemetry. (visited on 02-09-2024).

[66] Gnocchi. "Gnocchi." [Online]. Available: https://github.com/gnocchixyz/gnocchi. (visited on 02-09-2024).

[67] OpenStack. "Introduction to Octavia." [Online]. Available: https://docs.openstack.org/octavia/latest/reference/introduction.html. (visited on 03-09-2024).

[68] GeeksforGeeks. "What Is OpenStack Heat Service?" [Online]. Available: https://www.geeksforgeeks.org/what-is-openstack-heat-service/. (visited on 03-09-2024).

[69] OpenStack. "Heat." [Online]. Available: https://wiki.openstack.org/wiki/Heat. (visited on 03-09-2024).

[70] SlideShare. "OpenStack Heat." [Online]. Available: https://www.slideshare.net/slideshow/openstack-heat-51867693/51867693#3. (visited on 03-09-2024).

[71] OpenStack. "Senlin." [Online]. Available: https://wiki.openstack.org/wiki/Senlin. (visited on 03-09-2024).

[72] OpenStack. "Senlin Overview." [Online]. Available: https://docs.openstack.org/senlin/mitaka/overview.html. (visited on 03-09-2024).

[73] OpenStack. "Theory of Auto Scaling." [Online]. Available: https://docs.openstack.org/auto-scaling-sig/latest/theory-of-auto-scaling.html. (visited on 03-09-2024).

[74] New Relic. "What Is Prometheus?" [Online]. Available: https://newrelic.com/blog/best-practices/what-is-prometheus. (visited on 05-09-2024).

[75] Scale Your App. "What Is Grafana? Why Use It? Everything You Should Know About It." [Online]. Available: https://scaleyourapp.com/what-is-grafana-why-use-it-everything-you-should-know-about-it/. (visited on 05-09-2024).

[76] DevOps School. "What Is Prometheus and How It Works." [Online]. Available: https://www.devopsschool.com/blog/what-is-prometheus-and-how-it-works/. (visited on 05-09-2024).

[77] Guille, K. "Supervision of an OpenStack Infrastructure with Prometheus and Grafana." [Online]. Available: https://killianguille.wordpress.com/portfolio/supervision-of-an-openstack-infrastructure-with-prometheus-and-grafana/. (visited on 05-09-2024).

[78] OpenStack. "Heat Documentation." [Online]. Available: https://docs.openstack.org/heat/latest/. (accessed on 17-09-2024).

[79] Dake, S. "Heat API: OpenStack AWS CloudFormation Orchestration." [Online]. Available: https://github.com/sdake/slides/blob/master/sdake-cloudopen-heat-2012-08/heat-cloudopen-final.pdf. (accessed on 17-09-2024).

[80] https://ibm-blue-box-help.github.io/help-documentation/heat/autoscaling-with-heat/ (accessed on 17-09-2024).

[81] "VNF Scaling Operations with Heat." 2019. [Online]. Available: https://www.cloudqubes.com/hands-on/openstack/heat/vnf-scaling-operations-with-heat/

[82] "Autoscaling with Heat." 2019. [Online]. Available: https://opendev.org/openstack/senlin/src/branch/unmaintained/zed/README.rst. (accessed on 17-09-2024).

[83] "Auto-Scaling Your Apps with OpenStack Heat—An Orchestration Tool Comparison Pt I of II." 2015. [Online]. Available: https://cloudify.co/blog/openstack-summit-vancouver-cloud-network-orchestration-automation-heat-scaling/

[84] Kaur, K., Mangat, V., and Saluja, K.K. "A Study of OpenStack Networking and Auto-Scaling Using Heat Orchestration Template." In Intelligent Computing and Communication Systems, Springer, Berlin/Heidelberg, Germany, 2021, pp. 169–176.

[85] Gupta, P.R., Taneja, S., and Datt, A. "Using Heat and Ceilometer for Providing Autoscaling in OpenStack." International Journal of Information and Communication Technology, vol. 2, 2014, pp. 84–89.

[86] Yang, I., Tung, D.V., Kim, M., and Kim, Y. "Alarm-based Monitoring for a High Availability in Service Function Chain." In Proceedings of the 2016 International Conference on Cloud Computing, Research and Innovations (ICCCRI), Singapore, 4–5 May 2016, pp. 86–91.

[87] OpenStack. "Ceilometer Architecture." [Online]. Available: https://docs.openstack.org/ceilometer/mitaka/architecture.html. (accessed on 17-09-2024).

[88] Abaakouk, M. "Autoscaling with Heat and Ceilometer." Articles about OpenStack and Related Technologies from the RDO Community, 2013. [Online]. Available: https://blogs.rdoproject.org/2013/08/autoscaling-with-heat-and-ceilometer/. (accessed on 17-09-2024).

[89] Gomez-Rodriguez, M.A., Sosa-Sosa, V.J., and Gonzalez-Compean, J.L. "Assessment of Private Cloud Infrastructure Monitoring Tools: A Comparison of Ceilometer and Monasca." In Proceedings of the 6th International Conference on Data Science, Technology and Applications (DATA2017), Madrid, Spain, 24–26 July 2017, pp. 371–381.

[90] "Heat-Monasca-Auto-Scaling." [Online]. Available: https://wiki.openstack.org/wiki/Heat-Monasca-Auto-Scaling. (accessed on 17-09-2024).

[91] Lanciano, G., Galli, F., Cucinotta, T., Bacciu, D., and Passarella, A. "Predictive Auto-Scaling with OpenStack Monasca." In Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing (UCC ’21), Leicester, UK, 6–9 December 2021, pp. 1–10.

[92] Llorens-Carrodeguas, A.L., Leyva-Pupo, I., Cervello-Pastor, C., and Pineiro, L. "An SDN-based Solution for Horizontal Auto-Scaling and Load Balancing of Transparent VNF Clusters." Sensors, vol. 21, 2021, article 8283.

[93] Rahman, S., Ahmed, T., Huynh, M., Tornatore, M., and Mukherjee, B. "Auto-Scaling VNFs Using Machine Learning to Improve QoS and Reduce Cost." In Proceedings of the 2018 IEEE International Conference on Communications (ICC 2018), Kansas City, MO, USA, 20–24 May 2018, pp. 1–6.

[94] Arabnejad, H., Pahl, C., Jamshidi, P., and Estrada, G. "A Comparison of Reinforcement Learning Techniques for Fuzzy Cloud Auto-Scaling." In Proceedings of the 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), Madrid, Spain, 14–17 May 2017, pp. 64–73.

[95] Soto, P., Vleechauwer, D.D., Camelo, M., Bock, Y.D., Schepper, K.D., Chang, C.Y., Hellinckx, P., Botero, J.F., and Latre, S. "Towards Autonomous VNF Auto-Scaling Using Deep Reinforcement Learning." In Proceedings of the 2021 Eighth International Conference on Software Defined Systems (SDS), Gandia, Spain, 6–9 December 2021, pp. 1–8.

[96] Subramanya, T. and Riggio, R. "Centralized and Federated Learning for Predictive VNF Autoscaling in Multi-Domain 5G Networks and Beyond." IEEE Transactions on Network and Service Management, vol. 18, 2021, pp. 63–78.

[97] A. Ashraf, B. Byholm, and I. Porres, “Prediction-based VM provisioning and admission control for multi-tier web applications,” J. Cloud Comput., vol. 5, p. 15, 2016, doi: 10.1186/s13677-016-0065-9

[98] G. Quattrocchi, E. Incerto, R. Pinciroli, C. Trubiani and L. Baresi, "Autoscaling Solutions for Cloud Applications Under Dynamic Workloads" in IEEE Transactions on Services Computing, vol. 17, no. 03, pp. 804-820, May-June 2024, doi: 10.1109/TSC.2024.3354062.

[99] Amazon Web Services, “Step and simple scaling policies for Amazon EC2 Auto Scaling,” [Online]. Available: https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-simple-step.html. [Accessed: Dec. 19, 2024].

[100] Microsoft Azure, “AutoScale,” [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-overview#what-is-autoscale. [Accessed: Dec. 19, 2024].

[101] Amazon Web Services, “Target tracking scaling policies for Amazon EC2 Auto Scaling,” [Online]. Available:

https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-target-tracking.html. [Accessed: Dec. 20, 2024].

[102] Google Cloud Platform, “Autoscale to maintain a metric at a target value,” [Online]. Available: https://cloud.google.com/compute/docs/autoscaler/scaling-cloud-monitoring-metrics#configureutilizationtarget. [Accessed: Dec. 20, 2024].

[103] Amazon Web Services, “Scheduled Scaling for Amazon EC2 Auto Scaling,” [Online]. Available: https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-scheduled-scaling.html. [Accessed: Dec. 21, 2024].

[104] Microsoft Azure, “Azure AutoScale based on Schedule,” [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/tutorial-autoscale-performance-schedule#create-recurrence-profile. [Accessed: Dec. 21, 2024].

[105] Google Cloud Platform, “Schedules in GCP,” [Online]. Available: https://cloud.google.com/compute/docs/autoscaler#schedules. [Accessed: Dec. 21, 2024].

[106] Amazon Web Services, “Predictive Scaling for Amazon EC2 Auto Scaling,” [Online]. Available: https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-predictive-scaling.html. [Accessed: Dec. 22, 2024].

[107] Aqua Security, “Cloud workload security,” [Online]. Available: https://www.aquasec.com/cloud-native-academy/cspm/cloud-workload/. [Accessed: Dec. 21, 2024].

[108] R. Talwadker and C. George, "Yodea: Workload Pattern Assessment Tool for Cloud Migration," 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Nicosia, Cyprus, 2018, pp. 17-20, doi: 10.1109/CloudCom2018.2018.00019.

[109] V. Podolskiy, A. Jindal and M. Gerndt, "IaaS Reactive Autoscaling Performance Challenges," 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), San Francisco, CA, USA, 2018, pp. 954-957, doi: 10.1109/CLOUD.2018.00144.

[110] S. Patel, “Working with OpenStack Senlin cluster,” [Online]. Available: https://satishdotpatel.github.io/working-with-openstack-senlin-cluster/. [Accessed: Jan. 14, 2025].

[111] S. Patel, “OpenStack Senlin autoscaling,” [Online]. Available: https://satishdotpatel.github.io/openstack-senlin-autoscaling/. [Accessed: Jan. 14, 2025].

[112] D. K. T., “Auto-scaling OpenStack instances with Senlin and Prometheus,” [Online]. Available: https://medium.com/@dkt26111/auto-scaling-openstack-instances-with-senlin-and-prometheus-46100a9a14e1. [Accessed: Jan. 14, 2025].

[113] Red Hat, “What is YAML?” [Online]. Available: https://www.redhat.com/en/topics/automation/what-is-yaml. [Accessed: Jan. 15, 2025].

APPENDIX A

HEAT IMPLEMENTATION

```
heat_template_version: 2016-10-14
description: AutoScaling with Alarms
parameters:
  metadata:
    type: json
  flavor:
    type: string
    description: instance flavor to be used
    default: m1.nano
  network:
    type: string
    description: project network to attach instance to
    default: private
  security_group_id:
    type: string
    description: Security group used for the servers
    default: default
resources:
   instance_group:
    type: OS::Heat::AutoScalingGroup
    properties:
      cooldown: 180
      desired_capacity: 0
      max_size: 3
      min_size: 0
      resource:
        type: OS::Nova::Server
        properties:
       flavor: { get_param: flavor }
          image: cirros-0.5.2-x86_64-disk
          networks:
```

```
        - network: { get_param: network }
      security_groups:
        - get_param: security_group_id
      metadata: { "metering.server_group": { get_param: "OS::stack_id" } }

scaleout_policy:
  type: OS::Heat::ScalingPolicy
  properties:
    adjustment_type: change_in_capacity
    auto_scaling_group_id: { get_resource: instance_group }
    #cooldown: 120
    scaling_adjustment: 1

scalein_policy:
  type: OS::Heat::ScalingPolicy
  properties:
    adjustment_type: change_in_capacity
    auto_scaling_group_id: { get_resource: instance_group }
    #cooldown: 120
    scaling_adjustment: -1

cpu_alarm_high:
  type: OS::Aodh::GnocchiResourcesAlarm
  properties:
    description: Scale-up if the average cpu > 80% for 1 minute
    metric: cpu
    resource_type: instance
    resource_id: 3b372e3f-0a8f-4f5d-b5e6-e0226d672919
    enabled: true
    repeat_actions: true
    severity: critical
    granularity: 60
    aggregation_method: rate:mean
    evaluation_periods: 1
    threshold: 48000000000
```

```
      comparison_operator: gt
      alarm_actions:
        - str_replace:
            template: trust+url
            params:
              url: {get_attr: [scaleout_policy, alarm_url]}

  cpu_alarm_low:
    type: OS::Aodh::GnocchiResourcesAlarm
    properties:
      description: Scale-down if the average cpu > 10% for 1 minute
      metric: cpu
      resource_type: instance
      resource_id: 3b372e3f-0a8f-4f5d-b5e6-e0226d672919
      enabled: true
      repeat_actions: true
      severity: critical
      granularity: 60
      aggregation_method: rate:mean
      evaluation_periods: 1
      threshold: 36000000000
      comparison_operator: lt
      alarm_actions:
        - str_replace:
            template: trust+url
            params:
              url: {get_attr: [scalein_policy, alarm_url]}

outputs:
  scaleout_policy_signal_url:
    value: { get_attr: [scaleout_policy, alarm_url] }

  scalein_policy_signal_url:
    value: { get_attr: [scalein_policy, alarm_url] }
```

APPENDIX B

SENLIN IMPLEMENTATION

1. Create the configuration file to generate the cluster profile.

```
vim my_instance.yaml
type: os.nova.server
version: 1.0
properties:
name: my_instance
flavor: m1.nano
image: cirros 0.6.2
networks:
  - network: private
```

2. Create the cluster profile using the created configuration file.

- openstack cluster profile create --spec-file sample_server.yaml pserver

3. Create the cluster with the required capacity, min size, and max size using the cluster profile

- openstack cluster create --profile pserver --desired-capacity 0 --min-size 0 --max-size 3 mycluster
- export MYCLUSTER_ID=910e3e6f-07f6-4e42-b356-be482a204883

4. Create the cluster receiver to scale out the cluster once the High Alarm is detected.

- openstack cluster receiver create --action CLUSTER_SCALE_OUT --params count=1 --cluster mycluster r_01
- export ALRM_HIGH=http://192.168.204.130/cluster/v1/webhooks/b3482b7a-7ad0-456e-8111-26889edb5f2a/trigger?V=2&count=1

5. Create the cluster receiver to scale in the cluster once the Low Alarm is detected.

- openstack cluster receiver create --action CLUSTER_SCALE_IN --params count=1 --cluster mycluster r_02

- export ALRM_LOW=http://192.168.204.130/cluster/v1/webhooks/0a600322-2514-4697-ab04-57e68e05bf4d/trigger?V=2&count=1

5. Create a CPU High alarm to trigger once the CPU usage has reached 80%. The alarm action is to call the scale-out cluster receiver using the webhook.

```
openstack alarm create \
  --type gnocchi_resources_threshold \
  --name cpu-high \
  --metric cpu \
  --threshold 48000000000 \
  --comparison-operator gt \
  --description 'instance running hot' \
  --evaluation-periods 1 \
  --aggregation-method rate:mean \
  --alarm-action $ALRM_HIGH \
  --granularity 60 \
  --repeat-actions True \
  --resource-id 3b372e3f-0a8f-4f5d-b5e6-e0226d672919 \
  --resource-type instance \
  --query '{"=": {"id": "3b372e3f-0a8f-4f5d-b5e6-e0226d672919"}}'
```

6. Create a CPU Low alarm to trigger once the CPU usage has reduced to 60%. The alarm action is to call the scale-in cluster receiver using the webhook.

```
openstack alarm create \
  --type gnocchi_resources_threshold \
  --name cpu-low \
  --metric cpu \
  --threshold 36000000000 \
  --comparison-operator lt \
  --description 'instance running cold' \
  --evaluation-periods 1 \
  --aggregation-method rate:mean \
  --alarm-action $ALRM_LOW \
  --granularity 60 \
  --repeat-actions True \
  --resource-id 3b372e3f-0a8f-4f5d-b5e6-e0226d672919 \
  --resource-type instance \
  --query '{"=": {"id": "3b372e3f-0a8f-4f5d-b5e6-e0226d672919"}}'
```

7. Create a cluster policy configuration file, create a cluster policy using that file, and attach that policy to the above-created cluster for scale-out operation.

```
vim scaleout.yaml
type: senlin.policy.scaling
```

version: 1.0
description: Scaling policy with cooldown for scale-out
properties:
event: CLUSTER_SCALE_OUT
adjustment:
type: CHANGE_IN_CAPACITY
cooldown: 180

- openstack cluster policy create --spec-file scaleout.yaml scale_out_policy
- openstack cluster policy attach --policy scale_out_policy mycluster

8. Create a cluster policy configuration file, create a cluster policy using that file, and attach that policy to the above-created cluster for scale-in operation.

vim scalein.yaml
type: senlin.policy.scaling
version: 1.0
description: Scaling policy with cooldown for scale out
properties:
event: CLUSTER_SCALE_IN
adjustment:
type: CHANGE_IN_CAPACITY
cooldown: 180

- openstack cluster policy create --spec-file scalein.yaml scale_in_policy
- openstack cluster policy attach --policy scale_in_policy mycluster