ACTA SCIENTIAE
Revista de Ensino de Ciências e Matemática
ONLINE ISSN 2178-7727
ULBRA

SLA AWARE REACTIVE AUTOSCALING FOR CONTAINERIZED CLOUD APPLICATIONS USING APPLICATION AND INFRASTRUCTURE METRICS

Hirenkumar Ramanbhai Patel
Computer/IT Engineering, Gujarat Technological University, Chandkheda, Ahmedabad, Gujarat, India.
Tejas P. Patalia
Department Of Computer Engineering, V.V.P Engineering College, Rajkot, Gujarat, Ahmedabad

Abstract: Cloud computing is an online technology to provides computing resources (machines) to end-users on demand for running their applications over the internet. Applications hosted in a cloud computing environment may face fluctuating workloads. To deal with such fluctuating workloads cloud resources are allocated automatically to applications. Allocating cloud resources to applications in an automatic manner is known as Elasticity which can be implemented using auto-scaling. Auto-scaling can be implemented as a reactive or proactive approach. Cloud providers use Virtual Machine based or Container-based virtualization to host applications. Some of the factors that affect the availability of the application are computing resources and users accessing those applications. It is required to allocate/deallocate resources at the right moment, else failing to it can lead to SLA Violation which can result in cloud service user dissatisfaction, negative review for the cloud service provider, etc. During the literature study, it is found that reactive auto-scaling decisions are taken based on CPU utilization threshold. In this paper, we have proposed a reactive auto-scaling algorithm that uses application level (response time) and infrastructure level (CPU utilization) metrics together. This work has been evaluated and validated using our custom microservice-based application. The result shows that our approach improves 4% of SLA achievement and 3% in request processing during a simulation duration of 15 minutes.

Keywords: cloud computing, elasticity, auto-scaling, docker, container, reactive

1. INTRODUCTION

Cloud computing is an internet-based technology that provides various services to its users using service level agreement on the pay-per-use model. Infrastructure as a service (IaaS), Platform as a service (PaaS), and Software as services (SaaS) are three service models of cloud computing. Public, private, hybrid, or community cloud is known as deployment models [1].

In cloud computing, cloud service user pays an hourly or monthly fee for using software and computing resources. The cloud service user may allocate or deallocate (scale) the computing resource to achieve maximum application throughput using auto-scaling.

Using hardware virtualization, it is possible to run multiple virtual machines on the same physical machine while using operating system virtualization it is possible to run multiple containers on one virtual machine. The cloud service provider uses these virtualization technologies to provide computing services to cloud users. For hardware virtualization hypervisor is used and for operating system-level virtualization container is used[2].

A container contains an application with its dependencies. So, issues related to application deployment from a development environment to a real-time environment can be minimized. Application deployed in cloud environment needs computation resources for execution. Computation resources can be allocated by considering peak workload or average workload. If computation resources are allotted based on peak workload, then resources stay underutilized most of the time during the average workload. If computation resources are allotted based on the average workload then resources may be over-utilized during peak workload, which may result in resource wastage and SLA Violations [3]. So, cloud computing provides an elasticity feature to add or remove computing resources (like virtual machines or containers) automatically as per the application workload. Elasticity can be implemented using an auto-scaling mechanism[4].

Auto-scaling decisions can be performed dynamically. Such decisions depend on infrastructure-level metrics (such as average CPU utilization, and average memory utilization) or application-level metrics (such as average response time, throughput, etc.). Auto-scaling methods can be classified into two types namely reactive approach and proactive approach[5].

The reactive approach is purely based on a fixed threshold of infrastructure level or application-level metrics. For infrastructure-level, CPU Utilization, Memory utilization thresholds can be used and for application-level, application response time thresholds can be used[6]. The proactive approach is based on the prediction of incoming traffic or resource utilization in near future. In a proactive approach, resources can be added in advance to deal with SLA Violations or removed for better resource utilization [7]. Most cloud providers use a reactive approach to add/remove computing resources based on the Infrastructure level metric which is average CPU Utilization.

Elasticity can be classified as horizontal and vertical elasticity. In horizontal elasticity replica of a virtual machine or container with a similar configuration is added or removed while in vertical elasticity capacity of existing computing resources like a Virtual Machine or container is increased or decreased in terms of CPU core or memory[8].

In this work, we have proposed an approach for a reactive auto-scaling mechanism by considering both infrastructure-level metrics (average CPU Utilization) and application-level metrics (average response time) together. The scale-down decision is purely based on average response time.

A. Motivation

Containers are best suited for microservice-based applications. Compared with monolithic applications microservice-based applications gained much popularity recently. Initially, the cloud-hosted application may have limited users but after gaining some popularity application users may increase. As application users increase it is also required to add computing resources (containers) to handle the requests. So, to handle user requests it is required to add or remove containers dynamically. To maintain resources in a dynamic manner auto-scaling can be used.

2. BACKGROUND THEORY

A. Container

A container is an operating-system-level virtualization rather than hardware-level virtualization. It uses the kernel feature of the host operating system. The application can be hosted on a container with the help of the docker namespace. The container is used to run an application with its dependencies. containers have many advantages over the virtual machine as it is lightweight, starts within milliseconds and all containers share the same host OS kernel[9].

B. Docker

Docker architecture is based on a client-server model. Specifically, it is comprised of three components: Docker-daemon, Docker-client, and Docker-hub. Whenever a client sends a Docker command to the Docker daemon, the command is sent either through the command-line interface or via the Docker API. As a concept, both components can be found on the same host machine or a different host machine. In this article to simulate the proposed strategy, we used Docker API to scale in or scale out the containers[10].

C. Docker File

This is a text file that contains an instruction set complete with commands that you can use to create a Docker image. To run an application on the Docker container, there is an image that can be used for creating a Docker container and running the Docker container. It reads the instructions from the Docker file and creates the images automatically[11].

D. Docker Compose

Docker Compose is a command available in the docker tool it can be used to run multiple containers using the YML file (services.yml (Docker Documentation, n.d.)) which contains configuring instructions for one or more application services. It can start and stop all services using single command[12].

E. Docker Swarm

Docker swarm is a container orchestration tool. Using docker swam it is possible to create a cluster of virtual machines and run containers on it. Docker swarm is helpful for health check-ups of all the containers. It ensures that all the containers are running and if in case if any container or virtual machine fails it schedules the container on the available machine on the cluster. The tasks performed by the docker swarm is difficult to perform manually. In the docker swam environment one or more master nodes and multiple worker nodes available. Master nodes are used to control the operation of docker-swarm while worker nodes are used to host containers. Using docker-API container-related commands can be used like scaling up or down the containers[13].

F. Workload Generating Tool

We have used locust which is a modern and popular workload generating tool. Using locust, it is possible to define user behaviour with python code and a swarm system with millions of simultaneous users. Locust provides a web interface that provides options to enter the number of users, seconds to start users and the URL of the application. It also provides a command-line interface to use for generating workload for a fixed amount of time or fixed number of requests etc. Once the simulation starts, we may study the number of users, request rate, and response time using the chart provided by the locust web interface. It also contains a feature of downloading statistics[14].

3. RELATED WORK

In this section work related to container reactive auto-scaling has been presented.

M. C. De Abranches, P. Solis, and E. Alchieri [15] proposed an auto-scaling method for high-demanding web applications to increase efficiency in container allocation for processing web requests. Considered application response time threshold. used PID controller technique to calculate the required number of containers for a threshold of the desired average response time. Metrics used for the test are the average waiting time at the client application layer, the average waiting time at the HAProxy load balancer, the percentage of failed requests, the average number of containers allocated

during the test, and the average efficiency in container allocation. The result shows that the proposed method provides good results by allocating containers more efficiently.

S. Taherizadeh and V. Stankovski [16] proposed Dynamic multi-level auto-scaling rules for containerized applications. Considered infrastructure-level metrics like average CPU utilization and average memory utilization along with application-level metrics like service average response time. the proposed method was compared with seven existing auto-scaling methods in different synthetic and real-world workload scenarios. All auto-scalers were compared based on response time and the number of instantiated containers. The result shows that the proposed method has better overall performance under various types of workloads. Performance can be improved by considering application average response time during resource scale-down operation.

F. Zhang, X. Tang, X. Li, S. U. Khan, and Z. Li [17] proposed a container auto-scaler framework that monitors container resource utilization and scales in or scales out containers accordingly. They have carried out extensive experiments with different workload modes, workload durations, and scaling cool-down periods of times. Their experimental results show that the framework handles workload variation nicely with a short delay. They also observed that the repeat workload mode shows the best elasticity due to its recurring and predictable feature. Finally, they discover that the length of the cool-down period should be properly set up to balance system stability and good elasticity. They have used infrastructure-level metric CPU usage for scaling decisions.

Y. Al-Dhuraibi, F. Zalila, N. Djarallah, and P. Merle [18] presented the approach to coordinating the vertical elasticity of both Virtual machines and containers. They have proposed an auto-scaling technique for the applications to adjust resources at both container and VM levels according to the application workload with the help of an elastic controller. This controller modifies the container file system group for container vertical scaling. The scaling decision was taken based on infrastructure level metrics like average CPU utilization or memory utilization over a fixed interval of time using an upper threshold and lower threshold. This proposed auto-scaler outperforms the container vertical elasticity controller by 18.34 % and VM vertical elasticity controller by 70% and container horizontal elasticity by 39.6%.

P. Hoenisch, I. Weber, and S. Schulte [19] considered four dimensions of scaling for VMs and containers adjusting them horizontally (changes in the number of instances) and vertically(changes in the computation resources available to instances). They formulated the scaling decisions as multiple objective optimization problems. They evaluated their approach for realistic apps and concluded that their approach can reduce the average cost per request by about 20-20%. Used CPU utilization threshold.

Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle [20] proposed the first system powering vertical elasticity of docker containers named ElasticDocker which is rule-based and works autonomously. ElasticDocker was designed to scale up and down both CPU and memory assigned to each container as per the application workload. It adjusts memory, CPU time, and vCPU cores according to workload. It directly modifies the cgroup file system of docker contains to implement scaling decisions. live migration was also added as container resources are limited to VM resources. Experiments show that ElasticDeocker helps to reduce expenses for users, and better resource utilization for container providers. It improves QoS for application end-users. Elastic docker outperforms Kubernetes elasticity by 37.63%. infrastructure level metrics like average memory utilization and CPU utilization are used for scaling decisions.

E. Casalicchio and V. Perciballi [21] focused on selecting more appropriate performance metrics to activate auto-scaling actions. Investigated the use of relative and absolute metrics. relative metrics are based on the data collected from the /cgroup virtual file system using tools like docker stats or cAdvisor, the relative CPU utilization measure the share of CPU used by a container w.r.t other containers. Absolute metric values report the cumulative activity counter in the operating system. Results show that the use of absolute metrics is best suited for CPU intense workloads. A proposed new auto-scaling algorithm named KPH-A could reduce the application response time of a factor between 0.66 and 0.5 compared to Kubernetes horizontal auto-scaling algorithm. Used infrastructure-level metrics like relative and absolute CPU utilization and no application-level metrics.

M. S. Alexiou and E. G. M. Petrakis [13] proposed Elixir autonomous agent to extend the capabilities of the Docker swarm which supports the monitoring of resources in real-time. introduced a schedular that can do balancing the workload among swarm nodes, and supports auto-scaling of worker nodes. The proposed strategy is reactive and based on resource metrics like Memory, and CPU. The result shows that the proposed strategy responds to the increasing/decreasing resource demands of each application leading to a faster response time compared to the original non-auto-scaled version.

Xi Zheng et al [22] presented SmartVM which is an SLA-aware, microservice-centric deployment framework. That is designed to streamline the process of building and deploying dynamically scalable microservice which can handle traffic spikes in a cost-efficient manner. Considered resource utilization which is per container CPU percentage and memory usage and application-level metric SLA violation rate. Evaluation results show that the proposed approach advances in deployment cost, resource utilization, and SLA compliance also 66% cost reduction compared to the state-of-the-art uniform microservice approach with reduced SLA violation.

Satish Narayana [23] proposed a container-aware application scheduling strategy with an auto-scaling policy. The proposed strategy deploys the requested applications on the best-fit lightweight containers with minimum deployment time based on the resource requirements. Used bin packing strategy to deploy the application to a minimum number of the physical machine with efficient utilization of the computing resources. For auto-scaling used a heuristic-based policy for minimizing the wastage of the computing resources in the cloud data center. Used resource-level metrics like CPU and Memory utilization for scaling decisions. Compared proposed work using processing time, processing cost, resource utilization, and required number of PMS. The container-based auto-scaling strategy minimizes the 12-20% processing cost of the microservices and maximizes the CPU and memory utilization of the cloud servers by 9-15% and 10-18%, respectively, over the existing Docker Swarm strategies.

M. P. Yadav, H. A. Akarte, and D. K. Yadav [24] proposed container reactive autoscaling using a PID controller which takes response time as feedback and calculates the required number of containers to manage dynamic and fluctuating workload. The result shows improved performance of the system in terms of resource utilization and response time to manage fluctuating workload. The researcher only considered application-level metric average response time. The researcher suggested considering CPU and response time together for further improvements in resource utilization and application throughput.

Table 1 contains the summary of the rule-based reactive auto-scaling literature survey.

4. PROBLEM STATEMENT

Auto-scaling decisions are mostly based on Infrastructure level metric CPU Utilization or Memory Utilization. Resource utilization and application SLA Achievement can be improved if we consider the Application-level metric which is application response time along with the Infrastructure level metric.

Compared with the above-related work. The contribution of our proposed work includes the following:

- It is designed to maintain application response time within the defined SLA threshold.
- Better resource management while maintaining application response time.
- Considered both infrastructure level and application-level metrics for container scaling

5. PROPOSED SYSTEM

We have implemented a container auto scaler in the docker swarm environment to scale containers based on resource-level metrics (Average CPU Utilization) and Application-level metrics (Average response time). The overall idea is presented in Figure 1

Table 1. Literature Survey

Sr. No.	Work	Simulation Tool Name	Monitoring	Scaling Method
1	Marcelo cerqueira de Abranches et al.[15]	Kubernetes	Application	Horizontal
2	Salman taherizadeh et al.[16]	Kubernetes	CPU, Memory, Application	Horizontal
3	Xuxin Tang et al.[17]	DS/OS	CPU	Horizontal
4	Yahya Al-Dhuraibi et al.[18]	Docker Swarm	CPU, Memory	Vertical
5	TU Wien et al.[19]	-	CPU	Vertical & Horizontal
6	Yahya Al-Dhuraibi et al.[20]	Kubernetes	CPU, Memory	Vertical
7	Emiliano Casalicchio et al.[21]	Kubernetes	CPU	Horizontal
8	Michail S. Alexiou et al.[13]	Docker Swarm	CPU, Memory	Horizontal
9	Xi Zheng et al.[22]	Docker Swarm	CPU, Memory, Application	Horizontal
10	Satish Narayana et al.[23]	Docker Swarm	CPU, Memory	Horizontal
11	Mahendra Pratap Yadav et al.[24]	Response Time	Application	Horizontal

and the overall system architecture is given in figure 2. Reactive auto scaler checks for auto scaling condition in the fixed interval (30 seconds in our experiment) of time. In our experiment, 30 seconds is the cooldown period that avoids oscillations in container scaling.

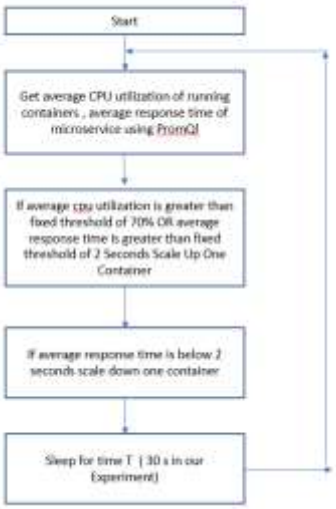

Figure 1 Proposed System

Auto scaler checks for average CPU utilization and average application response time. Auto scaler scales up the container if average CPU utilization is above 70% OR application average response time is above 2 seconds. Auto scaler scales down container if average response time stays below the threshold for a few minutes.

The system architecture consists of a locust (workload generator tool) (used separate system), one manager node, and four worker nodes. The manager node contains the Source of our custom microservice, Auto Scaler, Prometheus, and Grafana. Users can access microservice using any Node but in our experiment, we fixed access from Manager Node. Auto scaler is implemented using Docker SDK and Python programming language. It includes functions to get Container average CPU Utilization and Microservice average response time. Auto scaler also contains a data logger function to generate a log during simulation which contains time, average request rate, average response time, average CPU Utilization, and the number of containers running.

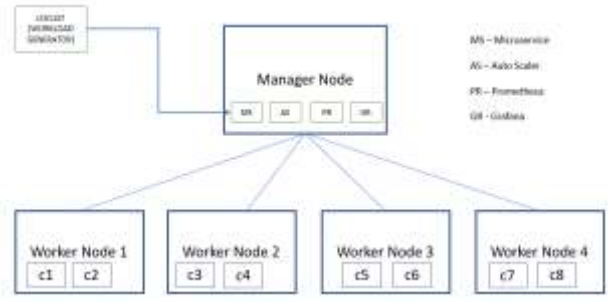

Figure 2 System Architecture

Prometheus is a monitoring and alerting toolkit used to record all the metrics during simulation. It is configured to fetch metrics in 15 seconds. Grafana is a data visualization tool. It allows to query, visualize and understand metrics no matter where they are stored. In our architecture, Grafana uses Prometheus as a data source and visualizes all major metrics to our custom-built dashboard. Worker nodes are part of the docker swarm. worker nodes are normally used to run

containers. For collecting various metrics data exporters are also required to run on worker nodes.

6. PERFORMANCE EVALUATION

A. Evaluation Environment Setting

a) Test Environment

The experiment was performed using 5 machines of Acer veriton installed ubuntu 18.04.6 LTS 64-bit operating system with memory 4 GB and AMD AB-6500 APU with radeon™ HD graphicsx4 processor. One machine was used as a swarm master node and the rest of the machines were used as worker nodes. The master node also installed monitoring tools like Prometheus version 2.0.0 and Grafana version 9.0.0 for metrics visualization. To monitor machine metrics used node exporter and for containers metrics used cadvisor.

We used 5 evaluation scenarios 1) no scaling 2) scaling using CPU utilization threshold only 3) scaling using response time threshold only 4) scaling using CPU or response time condition and 5) scaling using both CPU and response time condition. For all scenarios, we used our custom image of a container that runs the microservice-based application. Microservice accepts a number as a URL argument and generates prime numbers up to the passed number. Each container had processing and memory resources limited to 512 MB of Memory and a 40% share of the CPU.

b) Workload Generation

the workload generation was done using Locust which is an open-source load testing tool. Using locust, we can define user behaviour with python code and a swarm system with millions of simultaneous users.

In all scenarios, we generated synthetic workload using the locust workload generation interface. We evaluated all the scenarios for 1000 users, each user starts in a one-second interval and starts requesting the service. The synthetic load generated by Locust can be seen in figure 3.

c) Evaluation metrics

Our approach will be evaluated with metrics like % of request failure, SLA Achievement, Average Containers Used, and Average Response time.

Figure 3 The Locust tool generating synthetic workload

B. Evaluation results

For all scenarios, we configured a locust to generate 1 user every second until its count reaches 1000. For auto-scaling intervals, the cool-down period was fixed to 30 seconds. Max container limit was fixed to 16.

1) Scenario I: In this scenario, we used no container scaling to study how the system behaves.

Figure 4 shows the system behaviour if no container scaling is used. After 600 users system failed to handle requests and also response time reached 500,000 ms (8 minutes) and its shows 48% of request failures. Most of the time SLA is violated.

2) Scenario II: In this scenario, we used container auto-scaling based on the container average CPU utilization threshold to study how the system behaves.

Figure 5 shows the system behaviour of container auto-scaling based on the CPU utilization threshold. We can see a clear improvement in request handling and response time compared to the scenario of no container scaling. Requests starting failed after 600 users but failure is much lesser than no scaling scenario. Response time reached 400,000 (6 minutes) and request failure is only 15%.in this scenario SLA violations were less compared to no scaling scenario.

3) Scenario III: In this scenario, we used container auto-scaling based on the average response time threshold to study how the system behaves.

Figure 4 no of users, requests, and response time for scenario 1

Figure 5 No of users, requests, and response time for scenario 2

Figure 6 shows the system behaviour of container auto-scaling based on the microservice average response time threshold. We can see a clear improvement in response time. while users count was under 600, response time stays below

100,000 ms (1.6 minutes) compared to the scenario of auto-scaling based on container average CPU utilization. requests starting failed after 400 users' failure is much lesser than no scaling scenario but slightly more than average CPU utilization scenario its due to container limit of 16. Here requests failure is only 20%. With compared to CPU only scenario here most of the time response time stays within the threshold.

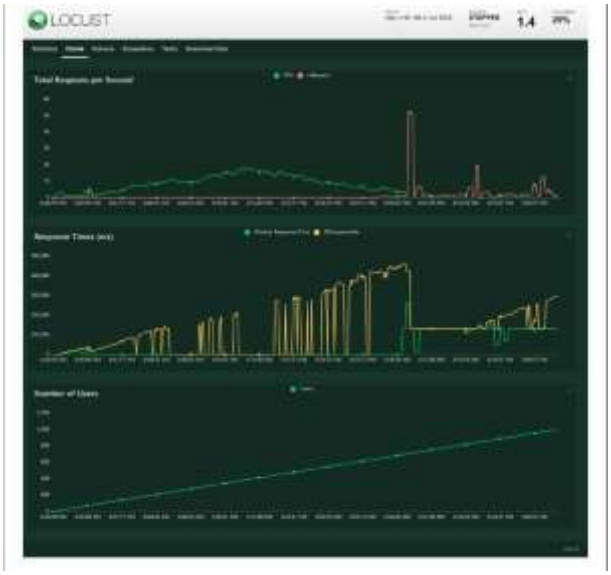

Figure 6 No of users, requests, and response time for scenario 3

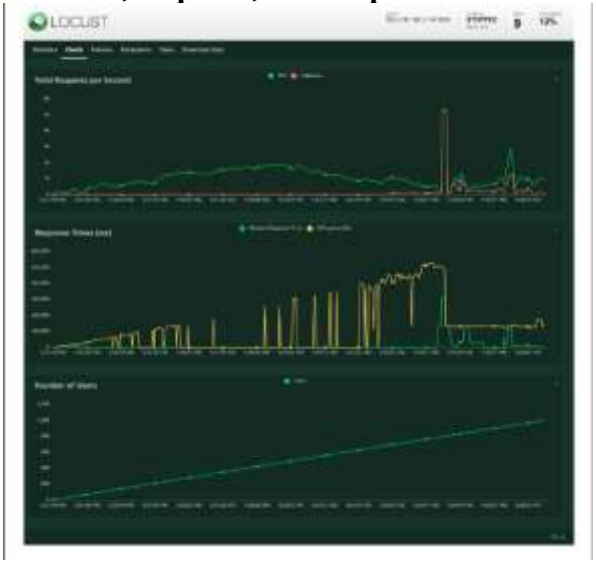

Figure 7 No of users, requests, and response time for scenario 4

4) Scenario IV: In this scenario, we used container auto-scaling based on the average response time threshold and container average CPU utilization together with OR conditions. To study how the system behaves.

Figure 7 shows the system behaviour of container auto-scaling based on microservice average response time and container average CPU utilization. We can see a clear improvement in response time compared to all the scenarios discussed previously. While user count was under 400, response time stays below 100,000 ms (1.6 minutes) compared to scenario other

scenarios. requests started failing after 800 users for a few minutes only and the system started handling requests with few failures. failure is much lesser than all the scenarios scenario. Here requests failure is only 12% only.

5) Scenario V: In this scenario, we used container auto-scaling based on average response time threshold and container average CPU utilization together but with AND condition. to study how the system behaves.

Figure 8 No of users, requests, and response time for scenario 5

Figure 8 shows the system behaviour of container auto-scaling based on microservice average response time threshold and container average CPU utilization. We can see the system can't handle requests and also result in SLA violations when the user count reaches 600. Here requests failure is only 36% which is higher than all scenarios except the scenario of no-scaling.

Figure 9 to 13 shows SLA Achievement graphs which were collected using Grafana tool using customized Prometheus promql queries. X-axes represent % of SLA achievements and Y axes represent the time duration of the simulation.

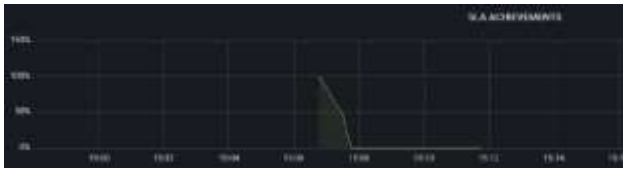

Figure 9 SLA Achievement Graph of Scenario 1

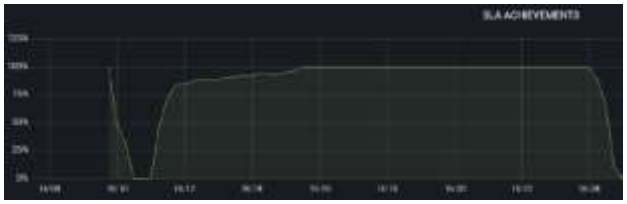

Figure 10 SLA Achievement Graph of Scenario 2

Figure 11 SLA Achievement Graph of Scenario 3

Figure 12 SLA Achievement Graph of Scenario 4

Figure 13 SLA Achievement Graph of Scenario 5

We have summarized the request failure and SLA achievement percentage of all scenarios in the below table.

Table 1. Requests failures % of all the scenarios for 1000 users.

Scenarios	% Of requests failures	SLA Achievement
No Scaling	48	0% most of the time
CPU Only	15	Between 75% - 100%
Response Time Only	20	Between 75% - 100%
CPU and Response time with OR Condition	12	Between 90%-100%
CPU and Response time with AND Condition	36	Below 80% most of the time

Table 2 data is represented in form of a graph in figure 14 for analysis. Compared to other scenarios, in scenario 4 we got an improvement in % of user requests that failed which is only 12 %.

Figure 14 % of User requests failed

Figure 15 represents the average number of containers used in each scenario. Scenario 4 it uses an average container of 7.1 which is required to keep response time within a threshold.

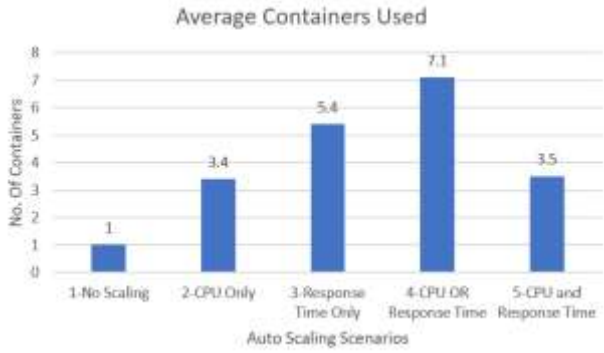

Figure 15 Average number of Containers used

Figure 16 represents the average response time measured for each scenario. scenario 3 has a good average response as compared with scenario 4.

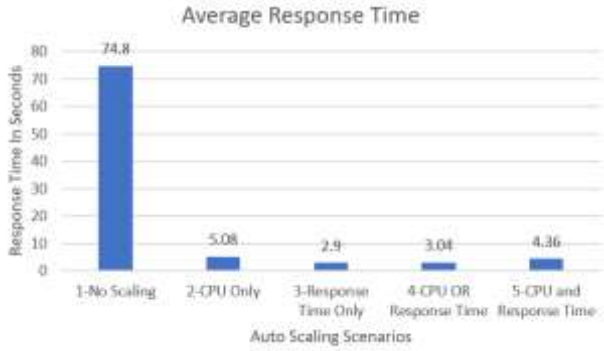

Figure 16 Average Response Time

7. CONCLUSION AND FUTURE WORK

In this work, we explored the use of application-level metrics (Response time) and Infrastructure level metrics (CPU Utilization) together for container scaling decisions. We experimented with five scenarios 1) No Scaling 2) scaling using CPU Only threshold 3) scaling using application response only threshold 4) scaling using both CPU utilization and application response time with OR Condition and 5) scaling using both CPU utilization and application response time with AND Condition.

The experiment result shows that scaling decisions based on both CPU threshold and average response time with OR Condition offers good efficiency and scalability for the system. The experiment results show improvement in terms of total requests handled and SLA achievement. Containers were gradually increased with the increase of response time or increase of CPU

utilization and similarly, containers were decreased gradually when response time was below the SLA threshold, we got better results compared to all other scenarios.

In current work we gradually increased or decreased the containers for scaling in the future we will use a mechanism to increase or decrease some multiple containers to immediately address the SLA violations and also for better resource usage in case of no violations. In the current work, we considered only one microservice-based application it will be an interesting future work to consider multiple microservice-based applications and to scale up only bottleneck/overutilized microservice. In the current work, we use the default load balancer of the docker swarm in future work we will use an external load balancer to study system behaviour. In our current work, we used the reactive approach in the future will use a proactive approach with a reactive approach.

REFERENCES

[1] H. Mezni, S. Aridhi, and A. Hadjali, "THE UNCERTAIN CLOUD: STATE OF THE ART AND RESEARCH CHALLENGES," *Int. J. Approx. Reason.*, vol. 1, pp. 1–14, 2018.

[2] M. K. Hussein, M. H. Mousa, and M. A. Alqarni, "A placement architecture for a container as a service (CaaS) in a cloud environment," *J. Cloud Comput.*, vol. 8, no. 1, 2019.

[3] M. P. Yadav, Rohit, and D. K. Yadav, "Maintaining container sustainability through machine learning," *Cluster Comput.*, vol. 24, no. 4, pp. 3725–3750, 2021.

[4] E. F. Coutinho, F. R. de Carvalho Sousa, P. A. L. Rego, D. G. Gomes, and J. N. de Souza, "Elasticity in cloud computing: a survey," *Ann. des Telecommun. Telecommun.*, vol. 70, no. 7–8, pp. 289–309, 2015.

[5] M. Abdullah, W. Iqbal, A. Erradi, and F. Bukhari, "Learning predictive autoscaling policies for cloud-hosted microservices using trace-driven modeling," in *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, 2019, vol. 2019-Decem, no. October, pp. 119–126.

[6] L. Zhang, Y. Zhang, P. Jamshidi, L. Xu, and C. Pahl, "Service workload patterns for Qos-driven cloud resource management," *J. Cloud Comput.*, vol. 4, no. 1, pp. 1–21, 2015.

[7] M. S. Aslanpour and S. E. Dashti, "Proactive Auto-Scaling Algorithm (PASA) for Cloud Application," *Int. J. Grid High Perform. Comput.*, vol. 9, no. 3, pp. 1–16, 2017.

[8] J. Herrera and G. Molto, "Towards Bio-inspired Auto-Scaling Algorithms: An Elasticity Approach for Container Orchestration Platforms," *IEEE Access*, vol. 8, no. 1, pp. 52139–52150, 2020.

[9] X. Wan, X. Guan, T. Wang, G. Bai, and B. Y. Choi, "Application deployment using Microservice and Docker containers: Framework and optimization," *J. Netw. Comput. Appl.*, vol. 119, no. July, pp. 97–109, 2018.

[10] R. Peinl, F. Holzschuher, and F. Pfitzer, "Docker Cluster Management for the Cloud - Survey Results and Own Solution," *J. Grid Comput.*, vol. 14, no. 2, pp. 265–282, 2016.

[11] P. Prakash and R. Suresh, "Comparative analysis on Docker and virtual machine in cloud computing," *Int. J. Pure Appl. Math.*, vol. 117, no. 7 Special Issue, pp. 175–184, 2017.

[12] M. P. Yadav, N. Pal, and D. K. Yadav, "Resource provisioning for containerized applications," *Cluster Comput.*, vol. 5, 2021.

[13] M. S. Alexiou and E. G. M. Petrakis, "Elixir: An Agent for Supporting Elasticity in Docker Swarm," *Adv. Intell. Syst. Comput.*, vol. 1151 AISC, pp. 1114–1125, 2020.

[14] N. C. Coulson, S. Sotiriadis, and N. Bessis, "Adaptive microservice scaling for elastic applications," *IEEE Internet Things J.*, pp. 1–1, 2020.
[15] M. C. De Abranches, P. Solis, and E. Alchieri, "PAS-CA : A Cloud Computing Auto-scalability Method for High-demand Web Systems," in *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, 2017, pp. 1–4.
[16] S. Taherizadeh and V. Stankovski, "Dynamic multi-level auto-scaling rules for containerized applications," *Comput. J.*, vol. 62, no. 2, pp. 174–197, Feb. 2019.
[17] F. Zhang, X. Tang, X. Li, S. U. Khan, and Z. Li, "Quantifying cloud elasticity with container-based autoscaling," *Futur. Gener. Comput. Syst.*, vol. 98, pp. 672–681, 2019.
[18] Y. Al-Dhuraibi, F. Zalila, N. Djarallah, and P. Merle, "Coordinating vertical elasticity of both containers and virtual machines," in *CLOSER 2018 - Proceedings of the 8th International Conference on Cloud Computing and Services Science*, 2018, vol. 2018-Janua, pp. 322–329.
[19] P. Hoenisch, I. Weber, and S. Schulte, "Four-fold Auto-scaling for Docker Containers," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 9435, no. November, pp. 0–8, 2015.
[20] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER," in *IEEE International Conference on Cloud Computing, CLOUD*, 2017, vol. 2017-June, pp. 472–479.
[21] E. Casalicchio and V. Perciballi, "Auto-Scaling of Containers: The Impact of Relative and Absolute Metrics," in *Proceedings - 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems, FAS*W 2017*, 2017, pp. 207–214.
[22] T. Zheng *et al.*, "SmartVM: a SLA-aware microservice deployment framework," *World Wide Web*, World Wide Web, pp. 1–19, 2018.
[23] S. N. Srirama, M. Adhikari, and S. Paul, "Application deployment using containers with auto-scaling for microservices in cloud environment," *J. Netw. Comput. Appl.*, vol. 160, no. February, p. 102629, 2020.
[24] M. P. Yadav, H. A. Akarte, and D. K. Yadav, "Container Elasticity: Based on Response Time using Docker," *Recent Adv. Comput. Sci. Commun.*, vol. 15, no. 5, pp. 773–785, 2020.