

Mitigating Auto-scaling Delays in Elastic-Docker for Better Responsiveness of Burst and Dynamic Workloads

Tarek Helmy

Information and Computer Science Department, Interdisciplinary Research Center for Intelligent Secure Systems,
King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia
Email: helmy@kfupm.edu.sa

Abstract—Dynamic assignment of resources to containers at runtime is crucial for improving users' quality of experience while maximizing resource utilization in a cloud environment. ElasticDocker is a threshold-based system for vertically auto-scaling Docker containers in the cloud. This paper presents a set of improvements aimed at improving the responsiveness of ElasticDocker to burst workloads. The objective involves generating predictions and assessing resource utilization for the original ElasticDocker implementation and the proposed solution. The analysis aims to provide valuable insights into resource utilization patterns and optimization opportunities. We compared both systems in the context of a video conversion workload. The economic feasibility and benefits of adopting the proposed improvements can be determined by comparing resource utilization and cost implications. The results show that the proposed improvements improve responsiveness in burst workloads allowing the system to finish the transcoding job 33.6% faster while consuming 6.2% less memory. We also investigate the impact of ElasticDocker with the proposed improvements on the performance and compare the results with the original ElasticDocker. The results show that when the number of requests increases and requires more resources, ElasticDocker with the proposed improvements reacts to provision resources accordingly, therefore the response time decreases, and the performance increases. The proposed set of improvements is simple yet effective and provides useful insights and a cornerstone in terms of how to start applying the elasticity of Docker containers for building a complete monitoring and adjusting system for running containers for researchers and practitioners.

Keywords—auto-scaling delays, elastic-docker, dynamic workloads

I. INTRODUCTION

One of the key advantages of cloud computing is elasticity. Many services experience fluctuating levels of demand throughout the day, and the ability to adjust resource allocation accordingly is crucial for efficient resource utilization and cost-effectiveness. By leveraging elasticity, service providers can effectively manage their

resources and provide high-quality services to their users while also keeping costs under control. Elasticity is defined as the ability of a service provider to scale their services on demand to meet client needs [1–3]. Rather than owning and managing physical computing resources such as servers, storage, and software applications, cloud computing enables users to access these resources as a service over the Internet. Cloud computing's Infrastructure-as-a-Service (IaaS) and other service models offer a significant advantage with their pay-as-you-go model, which can lead to cost optimization [3]. In cloud computing, this translates to either vertically scaling resources (CPU/memory/storage) on a single machine or horizontally provisioning more machines to meet increasing demand. That way multi-tenancy is enabled, where multiple clients can utilize the same machine without interfering with one another which maximizes the utilization of resources and minimizes costs for the clients because they only pay for the resources they consume.

Virtualization techniques are a cornerstone of elasticity as cloud providers nowadays offer access to virtualized resources instead of physical ones (e.g., CPU cores). It isolates tenants on the same hardware completely which in addition to maximizing resource utilization simplifies other aspects such as data backup and restoration. Furthermore, the advent of lightweight (OS-level) virtualization uses the host kernel to create isolated user spaces (containers) for applications to run [2]. A container packages an application with all its dependencies as a single software deployment unit. A container is usually crafted to support a single application, which enables developers to compose a set of containers to form a micro-service and deploy it on the cloud. This decouples components of a given service and allows the cloud provider to scale each of them individually on demand. This improves customer satisfaction through cost reduction and allows cloud providers to maximize resource utilization across their infrastructure. However, the process of scaling a container relies on its workload, thus containers with burst or dynamic workloads need more sophisticated approaches for anticipating it and scaling them accordingly. Therefore, researchers have proposed many techniques for vertically and horizontally scaling

containers to maximize resource utilization while minimizing Service Level Objective (SLO) violations. These techniques range from simple threshold-based ones as in [1] to more sophisticated predictive ones that use machine learning techniques as in [3, 4]. In this paper, we propose a set of improvements to the threshold-based system presented in [1]. Furthermore, we implement the improved scaling system and compare it with the original using a video conversion workload. Moreover, we use three metrics for comparing both systems that are suitable for our workload.

The rest of this paper is organized as follows: Section II introduces containers, Elastic-Docker and its resource management. Section III presents the literature review. Section IV presents the research method, the workload and the evaluation metrics used. Section V details the results and discussion. Finally, Section VI presents the conclusions and future work.

A. Containers

Containers are a lightweight and portable way to package and deploy applications, making it easy to run them on any platform that supports containers. Containers can be deployed quickly, can run on any infrastructure, and can be easily moved between different environments, such as development, testing, and production. Containers are a lightweight form of virtualization that allows applications to run in isolated environments with their dependencies. Unlike traditional Virtual Machines (VMs), which require a separate operating system for each instance, containers share the host operating system kernel. This shared kernel makes containers more lightweight, efficient, and portable than VMs. Containers encapsulate the application and its dependencies, such as the operating system, libraries, and runtime environment, providing a consistent and isolated runtime environment [4]. They offer a high level of resource efficiency, as multiple containers can run on a single host, utilizing the host's resources more effectively. In contrast, VMs emulate an entire computer system, including the operating system, and require a hypervisor to manage the virtualization layer. Each VM runs its operating system, which can result in higher resource usage and slower start-up times compared to containers. Containerized apps can be easily updated and scaled, as individual containers can be deployed or removed as needed without affecting the rest of the application. This approach enables developers to build, test, and deploy applications faster, with less effort and greater consistency across different environments. Containers can be allocated various types of resources to ensure optimal performance and resource utilization. These resources include CPU, memory, disk storage, network bandwidth, and I/O operations. The allocation of resources to containers can be specified through configuration settings or resource management tools. For example, CPU resources can be assigned by setting CPU limits or shares, which define the maximum amount of CPU time a container can consume. Memory resources can be allocated using limits and requests, specifying the maximum and minimum amount of memory a container requires. Disk storage can be assigned through volume mounts, allowing containers to

access specific directories or files. Network bandwidth can be controlled through network policies, ensuring containers have the necessary network resources. By accurately specifying resource allocations for containers, organizations can effectively manage and optimize resource usage, enabling efficient and reliable operation of containerized applications. Containerization is particularly useful for cloud-native applications, which are designed to run in cloud environments using a micro-services architecture.

Firstly, they enable easy deployment and management, allowing for independent scaling of each micro-service without affecting other services. This granular scalability ensures that resources can be efficiently allocated based on the specific demands of each micro-service. Additionally, containers facilitate seamless updates, as they can be replaced or modified without impacting the entire application. This flexibility enables agile development and continuous deployment practices, where new features or bug fixes can be rolled out quickly. Overall, containerization simplifies the management and scalability of micro-services in cloud environments, promoting efficient resource utilization and rapid application development.

B. Docker Overview

Docker is an OS-level virtualization technology that packages an entire application into a single deployable software unit with all its dependencies. It uses a client/server architecture, and it has the following main components:

- **Docker host:** Represents the host machine and manages Docker daemon, which manages all aspects of containers.
- **Docker client:** A user interface that manages all communication with the Docker daemon is done through the Docker client.
- **Docker registry:** Storage and distribution of containers.

Docker uses namespaces and control group features of the Linux/Unix kernel to create and manage containers. Namespaces are used to isolate (label) resources, while control groups (Cgroups) are used to manage those resources. Consequently, Cgroups isolate groups of processes (Containers) and resources are assigned to them using namespaces. These features are abstracted using container runtime (Runc), LXC, and libcontainer. Cgroups allow for managing CPU and memory, network, and disk bandwidth for each container. In addition, Cgroups expose many metrics about each container such as resource utilization levels. All those metrics are exposed in files on a pseudo-file system. For the Docker, these metrics are exposed at `"/sys/fs/cgroups/docker/containerid"`, where `"containerid"` is the id of the container that is under examination. Also, some of the files in this directory can be modified to adjust the limits of different resources assigned to this container with id `"containerid"`. These metrics can be exposed, and resource values can be adjusted through the Docker client. However, in this paper, we exclusively use the Docker client for updating resource

limits for containers and use the pseudo-file system to get up-to-date information on container resource utilization. The reason for that is the pseudo-file system is faster for getting new information, but it is not as safe when updating assigned resource values. Therefore, we use the Docker client for updating assigned resources, to ensure the reliable assignment of resources.

In this paper, we only consider CPU and memory in our scaling operations. For the CPU, we configure the quota and period of the Docker Completely Fair Scheduler (CFS). The period is the total amount of processing time (in microseconds) available in each CPU and the quota is the amount assigned to the container out of the period. It must be more than 0 and less than the number of processing units available in the machine. For the memory, the scaler assigns memory to the container directly out of the total available memory. In all cases, the Docker client ensures that newly assigned resources to a container are available and safely rejects any erroneous assignment operations (e.g., assignment of an unavailable RAM).

II. LITERATURE REVIEW

The authors in [1, 5] develop ElasticDocker, which is a threshold-based vertical auto-scaling technique for Docker containers in the cloud. Dogani *et al.* [6, 7] presented Auto-scaling techniques in container-based cloud and edge/fog computing. Zhang *et al.* [8] and Casalicchio [9] presented a rule-based auto-scaling method for containers in the cloud. Zhao *et al.* [10] proposed a technique for distributed redundancy scheduling for micro-service-based applications at the edge.

Proactive approaches anticipate dynamic workloads and de-provision resources accordingly before they occur, resulting in better overall performance. However, they usually consume more resources, because of their complexity. Gandhi *et al.* [11] used Kalman filters in proactive auto-scaling techniques that improve the performance of applications by learning their resource requirements. Al-Asaly *et al.* [12] proposed a deep learning-based resource usage prediction model for resource provisioning in an autonomic cloud computing environment. Bauer *et al.* [13] proposed a hybrid, proactive auto-scaling mechanism on a level-playing field. Wajahat *et al.* [14] presented a neural network-based auto-scaling method that learns container application workload patterns. Radhika *et al.* [15] used recurrent neural networks and ARIMA models in a predictive auto-scaling technique. Moreno-Vozmediano *et al.* [16] used a reinforcement learning-based technique that learns applications' resource usage patterns to predict future workloads.

Auto-scaling approaches in the literature, whether vertical or horizontal, can be categorized into non-predictive and predictive [17–28]. Non-predictive approaches usually rely on a set of predefined rules to take scale-up/down decisions based on several metrics such as response time or CPU utilization. However, proactive

techniques usually try to scale resources before changes in the workload, as opposed to reactive techniques.

The proposed system is an incremental development over ElasticDocker, therefore, it is categorized as non-predictive. However, the proposed system has some sort of intelligence built into it, unlike the other approaches in the same category. This intelligence is characterized by considering the previous behavior of the scaler to dynamically increase de/provisioned resource amounts at runtime. Furthermore, the proposed system attempts to make very short-term resource needs, unlike predictive approaches that attempt to invest more in obtaining the long-term resource needs of the container.

III. EXPERIMENTAL RESULT AND DISCUSSION

We use a dockerized version of a popular open-source video conversion tool called FFMPEG [23]. We use the tool to convert a 4k sample video¹. The details of the conversion process and the video are presented in Table I. The reason for choosing a video conversion workload is because of its aggressive nature concerning CPU and memory consumption. Once a conversion job starts FFMPEG uses as much of the available CPU as possible and just enough RAM to complete the conversion job.

TABLE I. DETAILS OF THE CONVERSION PROCESS AND THE VIDEO FILE USED

	Original	Target	Description
Encoding	Dolby Vision HEVC H265	MP4 AVC	A typical video downscaling workload, where the resolution of the video is reduced, and the format is changed to maximize compatibility and reduce size.
Format	MP4	MP4	
Resolution	3840×2160	1920×1080	
Framerate	59.94 FPS		
Length	75 s		
Size	272.2 MB	39.7 MB	

We use Memory & CPU utilization and the amount of time taken to finish the workload as evaluation metrics to evaluate our scaling solution and compare it to ElasticDocker. These metrics are not new; in fact, they have been used by the authors of ElasticDocker [1] to compare their solution to Docker (fixed resource assignment). We use Graylog [29], a powerful log management and analysis platform. We chose this application because it consumes a lot of resources. Since Graylog centralizes and aggregates all log files from different sources, it can suddenly get overloaded, and that requires a lot of attention from the providers to adjust resources according to the need, particularly at peak times. Graylog is widely implemented in the industry and it is based on four main components Graylog Server, Elasticsearch, MongoDB, and Web Interface. Graylog Server is a worker that receives and processes messages and communicates with all other components. Its performance is CPU-dependent. Elasticsearch is a search server that stores all of the logs/messages. It depends on the RAM, disk, and also CPU. MongoDB stores read-only

¹ <https://www.demolandia.net/downloads.html?id=2657643>

configuration and metadata and does not experience much load. Web Interface is the user interface. We run three containers, the first one is for the Graylog server version 2.0.0 and Web interface 2.0.0 while the second and third are for Elasticsearch version 2.3.3 and MongoDB version 3.2.6 respectively. We use Docker version 1.9.1 and Docker Compose version 1.7.1. Docker Compose is used to define and run the different components of Graylog in the containers. We also installed Ubuntu 14.04 and Httpperf [30] version 0.9.0-2build1 on the second VM. It has 2vCPUs with 4GB RAM. We also set scripts to send log messages and overload the Graylog server on this VM. The two VMs are on different VLANs. Httpperf generates requests to query the Graylog server.

A. Elastic-Docker

ElasticDocker [1] is a threshold-based vertical auto-scaling system for Docker containers. It dynamically scales CPU and memory for containers using a lower threshold and an upper one. The lower threshold is set at 70% and the upper threshold is set at 90%. Consequently, the auto-scaling is done as follows:

- If the utilization of the CPU increases above 90% then the amount of CPU assigned to that container is increased by 10%. For memory, the amount of memory is increased by 256 MB.
- If CPU utilization drops below 70% then the amount of CPU assigned to that container is decreased by 10%. For memory, the amount of memory is decreased by 128 MB.
- After a scale-up operation, a 10-s pause is taken, where no scaling decision is made for the scaled container.
- After a scale-down operation, a 20-s pause is taken, where no scaling decision is made for the scaled container.

The current utilization level of the CPU is taken by averaging 4 readings in a 16-s time window (a reading every 4 s). This is done because CPU utilization fluctuates a lot. ElasticDocker considers available resources when scaling a container.

B. Proposed Improvement

The first improvement is related to the pause that is taken after each scaling operation. The reason for having a pause after each scaling operation is to allow the containers to adjust to the new resource limits, which has the following implications:

- The scaler must run a separate thread for each container because a given container might need to be scaled while another is paused. This increases the number of resources needed by the scaler.
- The pause reduces the effectiveness of the scaler when two consecutive scaling operations are needed to satisfy container needs. The scaler makes a scaling decision every 26 s in this case.
- After a given high workload is finished, the scaler takes a lot of time to scale resources back down because it needs to do consecutive scaling-down

operations. The scaler decides every 36 s in this case.

We propose removing this pause period to overcome the limitations. We argue that having this pause period is not needed in the first place because of the following reasons:

- The scaler never scales RAM below its current utilization level. Therefore, a pause after scaling down the RAM is not needed.
- Scaling down the amount of CPU given to a container would throttle the container for that period, in the worst-case scenario. Therefore, a pause after scaling down the CPU is not needed.
- When the scaler gives more resources to a container, there is also no fear of destabilizing the container, because the container is given more resources.

By removing this pause, the new scaler takes a scaling decision every 4 s, 2 instead of the original 26 s after each scaling up and 36 s after each scaling down. This greatly improves the responsiveness of the scaler when addressing increasing demand and scaling back down when the workload is finished. Furthermore, a single thread can handle scaling operations for all containers on the host instead of having a separate thread for each container. The second improvement that we propose is related to consecutive scaling operations of the CPU when the first improvement is implemented. This is because the scaling decisions are made more frequently, i.e., one every 4 s. Therefore, to speed up the process of scaling up or down further, we double the resources for one time when a given container makes two consecutive scaling decisions of the same type and a third decision of the same type is made. At that moment, double the standard amounts in ElasticDocker are given/taken away once. This ensures adequate response in the case of burst workloads, while also guaranteeing proper resource utilization because the scaling decisions are done frequently. For example, in the worst-case scenario, our system scales a given container by giving it 20% more CPU when it does not need it, but this 20% would be taken back in just 8 s, however, if the container needed those resources then that could significantly improve the performance of the container (20%). The last improvement we propose relates to resource utilization by adjusting the thresholds to 80% and 95%, respectively. That makes the system more aggressive in taking away resources from the containers while also being more conservative when giving them resources. Under the original ElasticDocker, an improvement like that might have adverse effects on the performance of the container, mainly because of the pauses the scaler takes after each scaling operation. However, this effect can be greatly minimized because the pauses are removed in our system.

C. Elastic-Docker and Improved System Comparison

We proposed the following 3 improvements to ElasticDocker:

1. Remove pauses after each scaling operation.

2. Double the amount of resources in the third consecutive scaling operation of the same type once.
3. Adjust thresholds to 80% and 95%.

We compare the original ElasticDocker to the proposed system in the context of the workload and measure the metrics described in Section VI. For both scaling solutions, the Docker container running FFMPEG starts with 1 CPU and 1 GB of RAM. The details of the machine used in all the experiments are shown in Table II. The results from our experiment show that our enhancements significantly reduce the amount of time taken to finish the workload. Using ElasticDocker the video conversion finished in 567 s, while with our proposed improvements it took only 376 s (i.e., 33.7% less than ElasticDocker). This is mainly because the proposed improvements allow the system to provision resources faster than the original ElasticDocker, which provisions them slower because of the pauses after each scaling operation. Also, it doubles the number of resources provisioned during a streak, which translates well with workloads of this type. Furthermore, the results also show that our proposed improvements lead to better resource utilization because the scaler makes a scaling decision every 4 s while provisioning resources conservatively and de-provisioning them aggressively. Figs. 1 and 2, show the assigned CPU and RAM, respectively. For the CPU, the improved system was able to assign more of the available CPU quicker, which resulted in better overall performance. For the memory, the improved scaler was able to satisfy the memory needs (1.7 GB) of the workload earlier than ElasticDocker because of its rapid scaling behavior, while also managing to provision less memory because of its adjusted thresholds. ElasticDocker provisioned 2048 MB while our system provisioned 1920 MB (i.e., it is 6.2% better). We also investigate the impact of ElasticDocker with the proposed improvements on the performance and compare the results with the original ElasticDocker. The results show that when the number of requests increases and requires more resources, ElasticDocker with the proposed improvements reacts to provision resources accordingly, therefore the response time decreases, and the performance increases.

The details of the machine used in all the experiments are shown in Table II.

TABLE II. MACHINE SPECIFICATIONS

Key	Value
CPU	Intel Core i7 Quad-Core 4700MQ @ 2.1 GHz
RAM	16GB
Storage	128GB SATA3.0 SSD
OS	Ubuntu 20.04 64bit
Virtual Tech.	Docker version 19.03.13, build 4484c46d9d

We also investigate the impact of ElasticDocker with the proposed improvements on the performance and compare the results with the original ElasticDocker. Therefore, we run experiments to evaluate the performance of the Graylog application in two cases: i) with the original ElasticDocker, and ii) with ElasticDocker with the proposed improvements. We generate different

workloads using httpperf to query and search information from Graylog via its REST API. Fig. 3 shows the results of comparison experiments of average response time between ElasticDocker and ElasticDocker with the proposed improvements. As shown in Fig. 3, we have different request rates 10 req./sec., 20 req./sec., etc. The more the number of requests, the more the RT increases. When the number of requests is between 10 and 100 requests per second, there is no significant difference in average response time between the two cases. However, when the number of requests increases and requires more resources, ElasticDocker with the proposed improvements reacts to provision resources accordingly, therefore the response time decreases, and the performance increases.

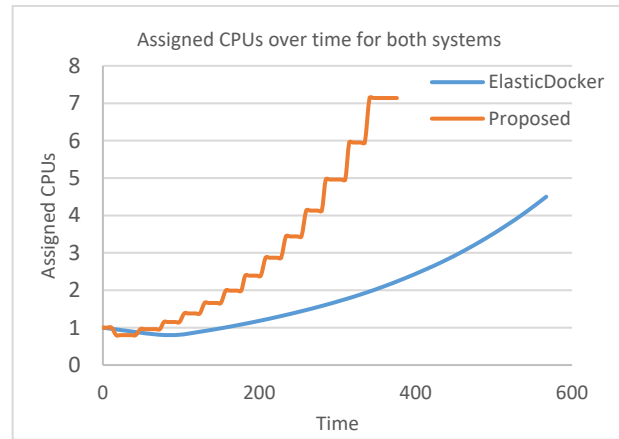


Fig. 1. Assigned CPU overtime for both systems

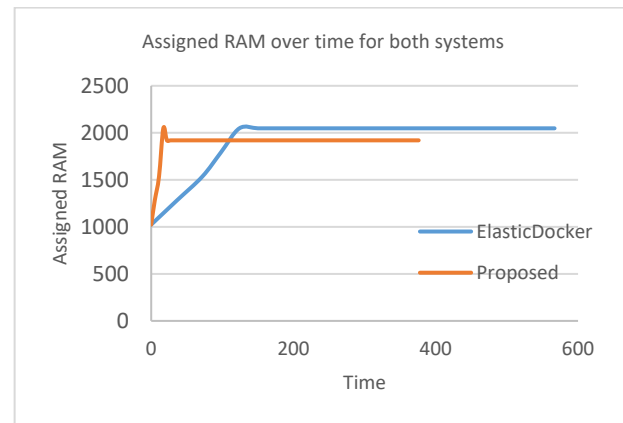


Fig. 2. Assigned RAM overtime for both systems.

D. Results and Discussion

The initial results show that our improved approach is better than ElasticDocker in terms of responsiveness to increasing demand for resources. However, this is one of the many workloads that could exist on a system, which might have been one of the causes that gave our system an edge over ElasticDocker. Another crucial aspect of the evaluation is the application that we used to evaluate both systems, which has no interactions with other processes or containers running on the system, unlike Graylog, which was used in ElasticDocker. Graylog runs through multiple containers that are orchestrated to work together (database, platform, web interface) unlike FFMPEG which runs in a

single container. The difference lies in the variability of resource requirements which might produce different results. However, our improvements are centered around improving reaction or response time to changing resource demand, which leads to the faster assignment of resources, in theory, but it still needs to be tested with more dynamic workloads. We also investigate the impact of ElasticDocker with the proposed improvements on the performance and compare the results with the original ElasticDocker. The results show that when the number of requests increases and requires more resources, ElasticDocker with the proposed improvements reacts to provision resources accordingly, therefore the RT decreases, and the performance increases. It is important to mention that the new scaler was not tested with different workloads. This is also a limitation of the original approach. However, in the case of our improved system, there is nothing to suggest that the system would not perform well under burst or dynamic workloads because scaling decisions are made every 4 s, which allows for quicker provisioning/de-provisioning of resources. The scaler was not tested in an environment with high resource contention (i.e., many containers competing for CPU and RAM), which might lead to a worse quality of experience for some of the running containers. However, this is a problem with all existing scalers of this type, where resources are assigned on a first come first serve basis and the scaler does not preempt resources from containers that need them.

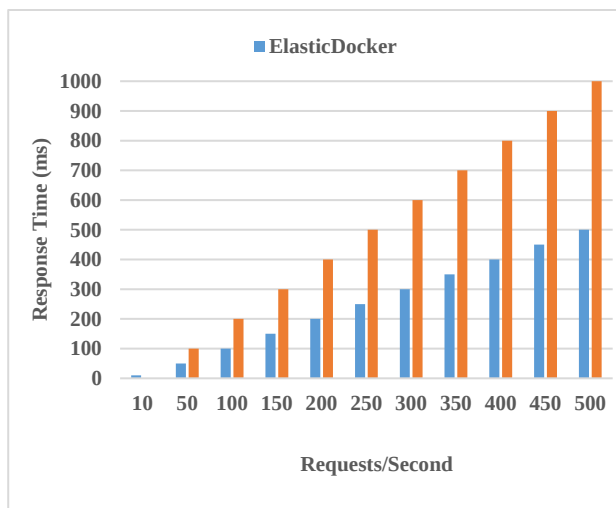


Fig. 3. Graylog response time with ElasticDocker and improved ElasticDocker.

IV. CONCLUSIONS AND FUTURE WORK

Auto-scaling solutions are used by cloud providers to help containerized applications cope with fluctuating workloads. Several industrial solutions have been proposed, ranging from ones based on simple rules to others that employ more sophisticated heuristics. In this paper, we proposed a set of improvements to ElasticDocker which aim to improve dynamic scaling behavior. The original systems and the proposed improvements were compared under a common video

conversion workload. Using a comprehensive empirical evaluation based on both simulated and cloud executions, we demonstrate how the proposed improvements outperform the original version under dynamic workloads. The results from the experiments show that the improved system led to better resource utilization, by quickly provisioning resources to the container, which led to finishing the workload 33.6% faster. Furthermore, results also showed that the new scaler was 6.2% better at provisioning memory, because it provisioned less memory, without adversely affecting the speed. We also investigate the impact of ElasticDocker with the proposed improvements on the performance and compare the results with the original ElasticDocker.

The results of the proposed improvements show that when the number of requests increases and requires more resources, ElasticDocker reacts to provision resources accordingly, therefore the response time decreases, and the performance increases. More improvements could be made in the future to address more limitations. First, implementing dynamic thresholds and scaling percentages determined by throttling statistics for containers could be used to address problems related to fairness and race condition resource assignment. Second, the use of time series prediction models to better guide scaling decisions, which are very attractive because they could be easily integrated, and they do not require as many resources as other machine learning techniques (e.g., Reinforcement Learning) at run time.

CONFLICT OF INTEREST

The author declares no conflict of interest.

ACKNOWLEDGMENT

The author acknowledges King Fahd University of Petroleum and Minerals and the Interdisciplinary Research Center for Intelligent Secure Systems for providing computing facilities and support for doing this work. Special thanks go to anonymous reviewers for their insightful comments and feedback, resulting in a significant improvement in the quality of the paper.

REFERENCES

- [1] J. Choi, M. Cho, and J.-S. Kim, "Employing vertical elasticity for efficient big data processing in container-based cloud environments," *Appl. Sci.*, vol. 11, no. 13, 6200, 2021. <https://doi.org/10.3390/app11136200>
- [2] Docker: Empowering App Development for Developers. [Online]. Available: <https://www.docker.com/>
- [3] K. Chengeta, "Comparing the performance between virtual machines and containers using deep learning credit models," in *Proc. the International Conference on Artificial Intelligence and its Applications*, 2021. <https://doi.org/10.1145/3487923.3487934>
- [4] M. Abdullah, W. Iqbal, J. L. Berral, J. Polo, and D. Carrera, "Burst-aware predictive auto-scaling for containerized microservices," *IEEE Transactions on Services Computing*, vol. 15, issue 3, pp. 1448–1460, May–June 2020.
- [5] N. Wang, M. Matthaoui, D. S. Nikolopoulos, and B. Varghese, "Dyverse: Dynamic vertical scaling in multi-tenant edge environments," *Future Generation Computer Systems*, vol. 108, pp. 598–612, July 2020.
- [6] What is a Container? [Online]. Available: <https://www.docker.com/resources/what-container/>

- [7] J. Dogani, R. Namvar, and F. Khunjush, "Auto-scaling techniques in container-based cloud and edge/fog computing taxonomy and survey," *Computer Communications*, vol. 209, no. 1, pp. 120–150, September 2023.
- [8] F. Zhang, X. Tang, X. Li, S. U. Khan, and Z. Li, "Quantifying cloud elasticity with container-based auto-scaling," *Future Generation Computer Systems*, vol. 98, pp. 672–681, September 2019.
- [9] E. Casalicchio, "A study on performance measures for auto-scaling CPU-intensive containerized applications," *Cluster Computing*, vol. 22, no. 3, pp. 995–1006, 2019.
- [10] H. Zhao, S. Deng, Z. Liu, J. Yin, and S. Dustdar, "Distributed redundancy scheduling for micro-service-based applications at the edge," *IEEE Trans. Serv. Comput.*, 2020. doi: 10.1109/TSC.2020.3013600
- [11] A. Gandhi, A. Karve, A. Kochut, and L. Zhang, "Providing performance guarantees for cloud-deployed applications," *IEEE Transactions on Cloud Computing*, pp. 269–281, vol. 8, 2020. doi: 10.1109/TCC.2017.2771402
- [12] M. S. Al-Asaly, M. A. Bencherif, A. Alsanad, and M. M. Hassan, "A deep learning-based resource usage prediction model for resource provisioning in an autonomic cloud computing environment," *Applications*, issue 13, pp 10211–10228, Jul. 2022. <https://doi.org/10.1007/s00521-021-06665-5>
- [13] A. Bauer, N. Herbst, S. Spinner, A. Ali-Eldin, and S. Kounev, "Chameleon: A hybrid, proactive auto-scaling mechanism on a level-playing field," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 800–813, 2019. doi: 10.1109/TPDS.2018.2870389
- [14] M. Wajahat, A. Karve, A. Kochut, and A. Gandhi, "Mlscale: A machine learning-based application-agnostic autoscaler," *Sustainable Computing: Informatics and Systems*, vol. 22, pp. 287–299, 2019.
- [15] E. G. Radhika, G. S. Sadasivam, and J. F. Naomi, "An efficient predictive technique to auto-scale the resources for web applications in the private cloud," in *Proc. 2018 Fourth International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics*, IEEE, 2018.
- [16] R. Moreno-Vozmediano, R. S. Montero, E. Huedo, and I. M. Llorente, "Efficient resource provisioning for elastic Cloud services based on machine learning techniques," *Journal of Cloud Computing*, vol. 8, 5, 2019.
- [17] Auto scale to maintain a metric at a target value. [Online]. Available: <https://cloud.google.com/compute/docs/autoscaler/scaling-cloud-monitoring-metrics#configure-utilization-target>
- [18] Kubernetes. Horizontal Pod Autoscaler. [Online]. Available: <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>
- [19] Google Kubernetes Engine. Vertical Pod Autoscaler. [Online]. Available: <https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler>
- [20] Amazon Web Services. Scheduled Scaling for Amazon EC2 Auto Scaling. [Online]. Available: <https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-autoscaling-scheduled-scaling.html>
- [21] Microsoft Azure. Azure Auto Scale based on Schedule. [Online]. Available: <https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/tutorial-autoscale-performance-schedule#create-recurrence-profile>
- [22] Google Cloud Platform. Schedules in GCP. [Online]. Available: <https://cloud.google.com/compute/docs/autoscaler#schedules>
- [23] A complete, cross-platform solution to record, convert and stream audio and video. [Online]. Available: <https://ffmpeg.org/>
- [24] Amazon Web Services. Predictive Scaling for Amazon EC2 Auto Scaling. [Online]. Available: <https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-autoscaling-predictive-scaling.html>
- [25] Y. Al-Dhuraibi, F. Zalila, N. Djarallah, and P. Merle, "Model-driven elasticity management with OCCI," *IEEE Trans. Cloud Comput.*, vol. 9, no. 4, pp. 1549–1562, 2021.
- [26] J.-Y. Choi, M. Cho, and J.-S. Kim, "Employing vertical elasticity for efficient big data processing in container-based cloud environments," *Applied Sciences*, vol. 11, no. 13, 6200, 2021. <https://doi.org/10.3390/app11136200>
- [27] Adaptive Resource Management Scheme in Container-Based Cloud Environments. [Online]. Available: <https://github.com/IDPL-MJU/ContainerVerticalElasticity/>
- [28] Runtime Options with Memory, CPUs, and GPUs. [Online]. Available: https://docs.docker.com/config/containers/resource_constraints/
- [29] Graylog 5.0 documentation. [Online]. Available: <https://go2docs.graylog.org/>
- [30] Httpperf tool. [Online]. Available: <https://en.wikipedia.org/wiki/Httpperf>

Copyright © 2024 by the authors. This is an open-access article distributed under the Creative Commons Attribution License (CC BY-NC-ND 4.0), which permits use, distribution and reproduction in any medium, provided that the article is properly cited, the use is non-commercial and no modifications or adaptations are made.